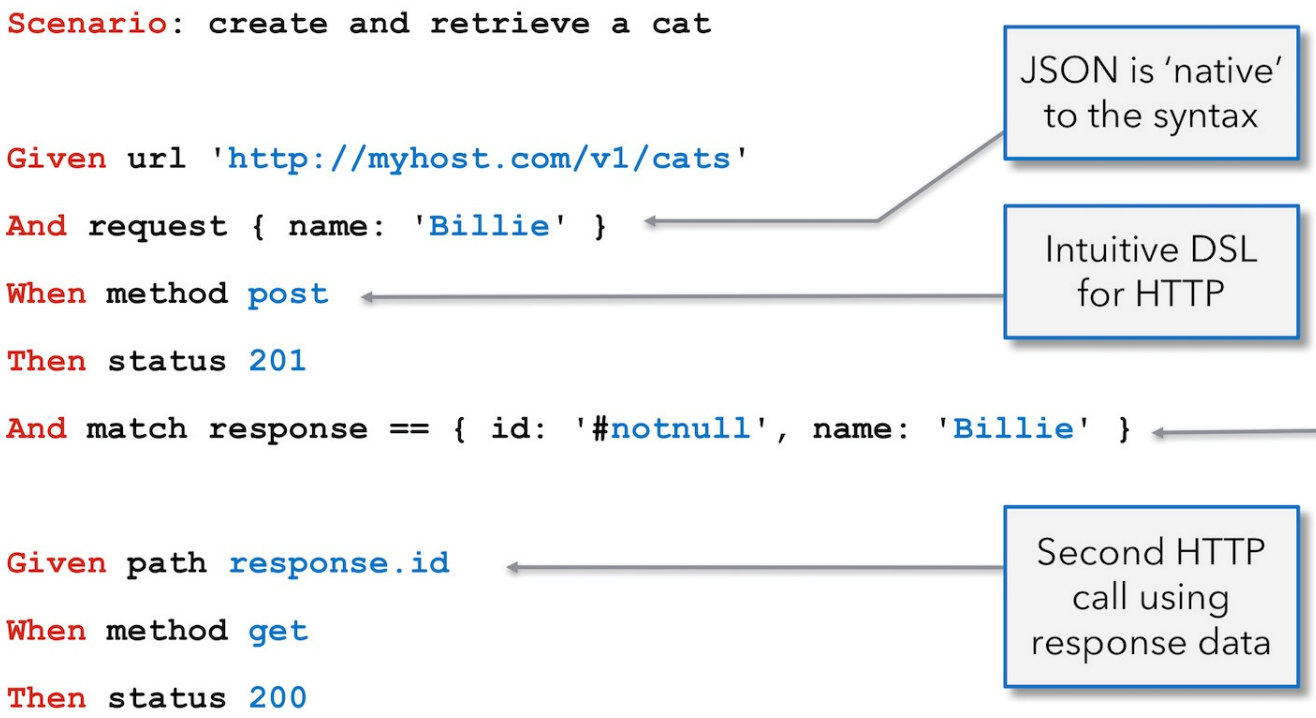
Karate

Web-Services Testing Made `Simple`.

Karate enables you to script a sequence of calls to any kind of web-service and assert that the responses are as expected. It makes it really easy to build complex request payloads, traverse data within the responses, and chain data from responses into the next request. Karate's payload validation engine can perform a 'smart compare' of two JSON or XML documents without being affected by white-space or the order in which data-elements actually appear, and you can opt to ignore fields that you choose.

Since Karate is built on top of Cucumber-JVM, you can run tests and generate reports like any standard Java project. But instead of Java - you write tests in a language designed to make dealing with HTTP, JSON or XML - **simple**.

Hello World



If you are familar with Cucumber, the *big difference* here is that you **don't** need to write extra "glue" code or Java "step definitions" !

It is worth pointing out that JSON is a 'first class citizen' of the syntax such that you can express payload and expected data without having to use double-quotes and without having to enclose JSON field names in quotes. There is no need to 'escape' characters like you would have had to in Java or other programming languages.

And you don't need to create Java objects (or POJO-s) for any of the payloads that you need to work with.

Index

		Functions		Conversion
Variables & Expressions	`def`	`assert` / `print` / `eval`	`text` / `replace`	`table` / `yaml`
Primary HTTP Keywords	`url`	`path`	`request`	`method`
....	`status`	`soap action`	`configure`	
Secondary HTTP Keywords	`param` / `params`	`header` / `headers`	`cookie` / `cookies`	`form field` / `form fields`
....	`multipart file` / `files`	`multipart field`	`multipart entity`	
Prepare, Mutate, Assert	`get` / `set` / `remove`	`match ==` / `!=`	`contains` / `only` / `any` / `!contains`	`match each`
Special Variables	`response`	`responseHeaders`	`responseCookies`	`responseStatus` / `responseTime`
Code Re-Use	`call` / `callonce`	Calling `*.feature` files	Calling JS Functions	Calling Java
Misc / Examples	Embedded Expressions	Polling / Conditional	XML and XPath	Tags / Grouping Tests
....	Data Driven Tests	Auth / Headers	Fuzzy Matching	Examples and Demos
....	Mock HTTP Servlet	Test Doubles / Contract Tests	Postman Import	Karate UI
....	Java API	Schema Validation	Karate vs REST-assured	Cucumber vs Karate

Features

- Java knowledge is not required and even non-programmers can write tests
- Scripts are plain-text files, require no compilation step or IDE, and teams can collaborate using standard version-control / Git
- Based on the popular Cucumber / Gherkin standard, and IDE support and syntax-coloring options exist
- Elegant DSL syntax 'natively' supports JSON and XML - including JsonPath and XPath expressions
- Eliminate the need for 'POJOs' or 'helper code' to represent payloads and HTTP end-points, and dramatically reduce the lines of code needed for a test
- Ideal for testing the highly dynamic responses from GraphQL API-s because of Karate's built-in text-manipulation and JsonPath capabilities

- Tests are super-readable - as scenario data can be expressed in-line, in human-friendly JSON, XML, Cucumber Scenario Outline tables, or a payload builder approach unique to Karate
- Express expected results as readable, well-formed JSON or XML, and assert in a single step that the entire response payload (no matter how complex or deeply nested) - is as expected
- Comprehensive assertion capabilities - and failures clearly report which data element (and path) is not as expected, for easy troubleshooting of even large payloads
- Embedded UI for stepping through a script in debug mode where you can even re-play a step while editing it - a huge time-saver
- Simpler and more powerful alternative to JSON-schema for validating payload structure and format that even supports cross-field / domain validation logic
- Scripts can call other scripts - which means that you can easily re-use and maintain authentication and 'set up' flows efficiently, across multiple tests
- Embedded JavaScript engine that allows you to build a library of re-usable functions that suit your specific environment or organization
- Re-use of payload-data and user-defined functions across tests is so easy - that it becomes a natural habit for the test-developer
- Built-in support for switching configuration across different environments (e.g. dev, QA, pre-prod)
- Support for data-driven tests and being able to tag or group tests is built-in, no need to rely on TestNG or JUnit
- Standard Java / Maven project structure, and seamless integration into CI / CD pipelines - with both JUnit and TestNG being supported
- Support for multi-threaded parallel execution, which is a huge time-saver, especially for HTTP integration tests
- Built-in test-reports powered by Cucumber-JVM with the option of using third-party (open-source) maven-plugins for even better-looking reports
- Reports include HTTP request and response logs in-line, which makes troubleshooting and debugging a test a lot easier
- Easily invoke JDK classes, Java libraries, or re-use custom Java code if needed, for ultimate extensibility
- Simple plug-in system for authentication and HTTP header management that will handle any complex, real-world scenario
- Future-proof 'pluggable' HTTP client abstraction supports both Apache and Jersey so that you can choose what works best in your project, and not be blocked by library or dependency conflicts
- API mock server for test-doubles that even maintain CRUD 'state' across multiple calls - enabling TDD for micro-services and Consumer Driven Contracts
- Mock HTTP Servlet that enables you to test **any** controller servlet such as Spring Boot / MVC or Jersey / JAX-RS - without having to boot an app-server, and you can use your HTTP integration tests un-changed
- Comprehensive support for different flavors of HTTP calls:
    - SOAP / XML requests
    - HTTPS / SSL - without needing certificates, key-stores or trust-stores
    - HTTP proxy server support

    - URL-encoded HTML-form data
    - Multi-part file-upload - including `multipart/mixed` and `multipart/related`
    - Browser-like cookie handling
    - Full control over HTTP headers, path and query parameters
    - Intelligent defaults

Real World Examples

A set of real-life examples can be found here: Karate Demos

Comparison with REST-assured

For teams familiar with or currently using REST-assured, this detailed comparison of Karate vs REST-assured - can help you evaluate Karate.

References

- REST API Testing with Karate - tutorial by Baeldung
- 10 API testing tools to try in 2017 - blog post by Christopher Reichert of Assertible
- Karate at the Ministry of Testing (Dallas) - presentation by Peter Thomas
- Testing a Java Spring Boot REST API with Karate - tutorial by Micha Kops - featured by Semaphore CI
- 5 top open-source API testing tools: How to choose - TechBeacon article by Joe Colantonio

You can find a lot more at the community wiki. Karate also has its own 'tag' and a healthy presence on Stack Overflow.

Getting Started

Karate requires Java 8 (at least version 1.8.0_112 or greater) and then either Maven, Gradle or Eclipse to be installed.

Maven

Karate is designed so that you can choose between the Apache or Jersey HTTP client implementations.

So you need two `<dependencies>`:

```
<dependency>
    <groupId>com.intuit.karate</groupId>
    <artifactId>karate-apache</artifactId>
    <version>0.7.0</version>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.intuit.karate</groupId>
    <artifactId>karate-junit4</artifactId>
    <version>0.7.0</version>
    <scope>test</scope>
</dependency>
```

And if you run into class-loading conflicts, for example if an older version of the Apache libraries are being used within your project - then use `karate-jersey` instead of `karate-apache`.

Gradle

Alternatively for Gradle you need two `dependencies`:

```
    testCompile 'com.intuit.karate:karate-junit4:0.7.0'
    testCompile 'com.intuit.karate:karate-apache:0.7.0'
```

TestNG instead of JUnit

If you want to use TestNG, use the artifactId `karate-testng`. If you are starting a project from scratch, we strongly recommend that you use JUnit. Do note that dynamic tables, data-driven testing and tag-groups are built-in to Karate, so that you don't need to depend on things like the TestNG `@DataProvider` anymore.

Use the TestNG test-runner only when you are trying to add Karate tests side-by-side with an existing set of TestNG test-classes, possibly as a migration strategy. One of the things you would miss if you use TestNG is the JUnit HTML report that is very useful for troubleshooting tests in dev-mode.

Quickstart

It may be easier for you to use the Karate Maven archetype to create a skeleton project with one command. You can then skip the next few sections, as the `pom.xml`, recommended directory structure and starter files would be created for you.

If you are behind a corporate proxy, or especially if your local Maven installation has been configured to point to a repository within your local network, the command below may not work. One workaround is to temporarily disable or rename your Maven `settings.xml` file, and try again.

You can replace the values of `com.mycompany` and `myproject` as per your needs.

```
mvn archetype:generate \
-DarchetypeGroupId=com.intuit.karate \
-DarchetypeArtifactId=karate-archetype \
```

```
-DarchetypeVersion=0.7.0 \
-DgroupId=com.mycompany \
-DartifactId=myproject
```

This will create a folder called `myproject` (or whatever you set the name to).

Eclipse Quickstart

You can refer to this nice blog post and video by Joe Colantonio which provides step by step instructions on how to get started using Eclipse (without having to run the command above). Use the latest available version of Karate (refer to the `archetypeVersion` above), and also make sure you install the Cucumber-Eclipse plugin !

Another blog post which is a good step-by-step reference is this one by Micha Kops - especially if you use the 'default' maven folder structure instead of the one recommended below.

Folder Structure

A Karate test script has the file extension `.feature` which is the standard followed by Cucumber. You are free to organize your files using regular Java package conventions.

The Maven tradition is to have non-Java source files in a separate `src/test/resources` folder structure - but we recommend that you keep them side-by-side with your `*.java` files. When you have a large and complex project, you will end up with a few data files (e.g. `*.js`, `*.json`, `*.txt`) as well and it is much more convenient to see the `*.java` and `*.feature` files and all related artifacts in the same place.

This can be easily achieved with the following tweak to your maven `<build>` section.

```
<build>
    <testResources>
        <testResource>
            <directory>src/test/java</directory>
            <excludes>
                <exclude>**/*.java</exclude>
            </excludes>
        </testResource>
    </testResources>
    <plugins>
    ...
    </plugins>
</build>
```

This is very common in the world of Maven users and keep in mind that these are tests and not production code.

Alternatively, if using gradle then add the following `sourceSets` definition

```
sourceSets {
    test {
        resources {
            srcDir file('src/test/java')
            exclude '**/*.java'
        }
    }
}
```

With the above in place, you don't have to keep switching between your `src/test/java` and `src/test/resources` folders, you can have all your test-code and artifacts under `src/test/java` and everything will work as expected.

Once you get used to this, you may even start wondering why projects need a `src/test/resources` folder at all !

Naming Conventions

Since these are tests and not production Java code, you don't need to be bound by the `com.mycompany.foo.bar` convention and the un-necessary explosion of sub-folders that ensues. We suggest that you have a folder hierarchy only one or two levels deep - where the folder names clearly identify which 'resource', 'entity' or API is the web-service under test.

For example:

```
src/test/java
    |
    +-- karate-config.js
    +-- logback-test.xml
    +-- some-reusable.feature
    +-- some-classpath-function.js
    +-- some-classpath-payload.json
    |
    \-- animals
        |
        +-- AnimalsTest.java
        |
        +-- cats
        |   |
        |   +-- cats-post.feature
        |   +-- cats-get.feature
        |   +-- cat.json
        |   \-- CatsRunner.java
        |
        \-- dogs
            |
            +-- dog-crud.feature
            +-- dog.json
            +-- some-helper-function.js
            \-- DogsRunner.java
```

Assuming you use JUnit, there are some good reasons for the recommended (best practice) naming convention and choice of file-placement shown above:

- Not using the `*Test.java` convention for the JUnit classes (e.g. `CatsRunner.java`) in the `cats` and `dogs` folder ensures that these tests will **not** be picked up when invoking `mvn test` (for the whole project) from the command line. But you can still invoke these tests from the IDE, which is convenient when in development mode.
- `AnimalsTest.java` (the only file that follows the `*Test.java` naming convention) acts as the 'test suite' for the entire project. By default, Karate will load all `*.feature` files from sub-directories as well. But since `some-reusable.feature` is *above* `AnimalsTest.java` in the folder hierarchy, it will **not** be picked-up. Which is exactly what we want, because `some-reusable.feature` is designed to be called only from one of the other test scripts (perhaps with some parameters being passed). You can also use tags to skip files.
- `some-classpath-function.js` and `some-classpath-payload.js` are in the 'root' of the Java 'classpath' which means they can be easily read (and re-used) from any test-script by using the `classpath:` prefix, for e.g: `read('classpath:some-classpath-function.js')`. Relative paths will also work.

For details on what actually goes into a script or `*.feature` file, refer to the syntax guide.

IDE Support

Many popular text editors such as Visual Studio Code have support for the Gherkin syntax. Using a Java IDE with Cucumber-JVM support is recommended for the best developer experience.

A debug helper was introduced in version 0.6.0 - which helps you set conditional break-points while running tests. Also refer to the JUnit HTML report.

Running in Eclipse or IntelliJ

If you use the open-source Eclipse Java IDE, you should consider installing the free Cucumber-Eclipse plugin. It provides syntax coloring, and the best part is that you can 'right-click' and run Karate test scripts without needing to write a single line of Java code.

If you use IntelliJ, Cucumber support is built-in and you can even select a single `Scenario` within a `Feature` to run at a time.

Troubleshooting Cucumber IDE Support

- On Eclipse you may see warnings such as `Step 'xxx' does not have a matching glue code` or `required(..)+ loop did not match anything at input Scenario:`, and on IntelliJ: `Unimplemented substep definition`. Refer to this ticket on how to solve this.

- On IntelliJ you may run into issues if JavaFX is not installed by default (e.g. on Ubuntu). Refer to this ticket for solutions.

`file.encoding`

In some cases, for large payloads and especially when the default system encoding is not `UTF-8` (Windows or non-US locales), you may run into issues where a `java.io.ByteArrayInputStream` is encountered instead of a string. Other errors could be a `java.net.URISyntaxException` and `match` not working as expected because of special or foreign characters, e.g. German or `ISO-8859-15`. Typical symptoms are your tests working fine via the IDE but not when running via Maven or Gradle. The solution is to ensure that when Karate tests run, the JVM `file.encoding` is set to `UTF-8`. This can be done via the `maven-surefire-plugin` configuration. Add the plugin to the `<build>/<plugins>` section of your `pom.xml` if not already present:

```
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.10</version>
        <configuration>
            <argLine>-Dfile.encoding=UTF-8</argLine>
        </configuration>
    </plugin>
```

Running With JUnit

To run a script `*.feature` file from your Java IDE, you just need the following empty test-class in the same package. The name of the class doesn't matter, and it will automatically run any `*.feature` file in the same package. This comes in useful because depending on how you organize your files and folders - you can have multiple feature files executed by a single JUnit test-class.

```
package animals.cats;

import com.intuit.karate.junit4.Karate;
import org.junit.runner.RunWith;

@RunWith(Karate.class)
public class CatsRunner {

}
```

Refer to your IDE documentation for how to run a JUnit class. Typically right-clicking on the file in the project browser or even within the editor view would bring up the "Run as JUnit Test" menu option.

Karate will traverse sub-directories and look for `*.feature` files. For example if you have the JUnit class in the `com.mycompany` package, `*.feature` files in `com.mycompany.foo` and

`com.mycompany.bar` will also be run. This is one reason why you may want to prefer a 'flat' directory structure as explained above.

JUnit HTML report

When you use the `@RunWith(Karate.class)` - after the execution of each feature, an HTML report is output to the `target/surefire-reports` folder and the full path will be printed to the console (see video).

```
html report: (paste into browser to view)
-------------------------------------------
file:/projects/myproject/target/surefire-reports/TEST-
mypackage.myfeature.html
```

You can easily select (double-click), copy and paste this `file:` URL into your browser address bar. This report is useful for troubleshooting and debugging a test because all requests and responses are shown in-line with the steps, along with error messages and the output of `print` statements. Just re-fresh your browser window if you re-run the test.

Running With TestNG

You extend a class from the `karate-testng` Maven artifact like so. All other behavior is the same as if using JUnit.

```
package animals.cats;

import com.intuit.karate.testng.KarateRunner;

public class CatsRunner extends KarateRunner {

}
```

Cucumber Options

To run only a specific feature file from a JUnit test even if there are multiple `*.feature` files in the same folder (or sub-folders), use the `@CucumberOptions` annotation.

```
package animals.cats;

import com.intuit.karate.junit4.Karate;
import cucumber.api.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Karate.class)
@CucumberOptions(features = "classpath:animals/cats/cats-post.feature")
public class CatsPostRunner {

}
```

The `features` parameter in the annotation can take an array, so if you wanted to associate multiple feature files with a JUnit test, you could do this:

```
@CucumberOptions(features = {
    "classpath:animals/cats/cats-post.feature",
    "classpath:animals/cats/cats-get.feature"})
```

And most convenient of all, you can even point to a directory (or package). Combine this with tags to execute multiple features, without having to list every one of them.

```
@CucumberOptions(features = "classpath:animals/cats", tags = "~@ignore")
// this will run all feature files in 'animals/cats'
// except the ones tagged as @ignore
```

Note that any `plugins` specified on the `@CucumberOptions` annotation will be ignored when using `@RunWith(Karate.class)`, because Karate's execution life-cycle is not compatible with 'native' Cucumber reports. This is not a limitation of Karate at all - as for API tests, you will only ever need the parallel report (which also produces standard JUnit and Cucumber-JSON output) or the HTML report.

For TestNG, the `@CucumberOptions` annotation can be used the same way as for JUnit.

Command Line

Normally in dev mode, you will use your IDE to run a `*.feature` file directly or via the companion 'runner' JUnit Java class. When you have a 'runner' class in place, it would be possible to run it from the command-line as well.

Note that the `mvn test` command only runs test classes that follow the `*Test.java` naming convention by default. But you can choose a single test to run like this:

```
mvn test -Dtest=CatsRunner
```

For gradle you must extend the test task to allow the `cucumber.options` to be passed to the Cucumber-JVM (otherwise they get consumed by gradle itself). To do that, add the following:

```
test {
    // pull cucumber options into the cucumber jvm
    systemProperty "cucumber.options",
System.properties.getProperty("cucumber.options")
    // pull karate options into the jvm
    systemProperty "karate.env", System.properties.getProperty("karate.env")
    // ensure tests are always run
    outputs.upToDateWhen { false }
}
```

And then the above command in gradle would look like:

```
./gradlew test -Dtest=CatsRunner
```

Test Suites

The recommended way to define and run test-suites and reporting in Karate is to use the parallel runner, described in the next section. The approach in this section is more suited for troubleshooting in dev-mode, using your IDE.

One way to define 'test-suites' in Karate is to have a JUnit class with the `@RunWith(Karate.class)` annotation at a level 'above' (in terms of folder hierarchy) all the `*.feature` files in your project. So if you take the previous folder structure example, you can do this on the command-line:

```
mvn test -Dcucumber.options="--tags ~@ignore" -Dtest=AnimalsTest
```

Here, `AnimalsTest` is the name of the Java class we designated to run the multiple `*.feature` files that make up your test-suite. Cucumber has a neat way to tag your tests and the above example demonstrates how to run all tests *except* the ones tagged `@ignore`.

The tag options can be specified in the test-class via the `@CucumberOptions` annotation, in which case you don't need to pass the `-Dcucumber.options` on the command-line:

```
@CucumberOptions(tags = {"~@ignore"})
```

You can 'lock down' the fact that you only want to execute the single JUnit class that functions as a test-suite - by using the following maven-surefire-plugin configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.10</version>
    <configuration>
        <includes>
            <include>animals/AnimalsTest.java</include>
        </includes>
        <systemProperties>
            <cucumber.options>--tags ~@ignore</cucumber.options>
        </systemProperties>
    </configuration>
</plugin>
```

Note how the `cucumber.options` can be specified using the `<systemProperties>` configuration. Options here would over-ride corresponding options specified if a `@CucumberOptions` annotation is present (on `AnimalsTest.java`).

For Gradle, you simply specify the test which is to be `include`-d:

```
test {
    include 'animals/AnimalsTest.java'
    // pull cucumber options into the cucumber jvm
```

```
    systemProperty "cucumber.options",
System.properties.getProperty("cucumber.options")
    // pull karate options into the jvm
    systemProperty "karate.env", System.properties.getProperty("karate.env")
    // ensure tests are always run
    outputs.upToDateWhen { false }
}
```

The big drawback of the 'Cucumber-native' approach is that you cannot run tests in parallel. The recommended approach for Karate reporting in a Continuous Integration set-up is described in the next section which focuses on generating the JUnit XML format that most CI tools can consume. The Cucumber JSON format is also emitted, which gives you plenty of options for generating pretty reports using third-party maven plugins.

And most importantly - you can run tests in parallel without having to depend on third-party hacks that introduce code-generation and config 'bloat' into your `pom.xml` or `build.gradle`.

Parallel Execution

Karate can run tests in parallel, and dramatically cut down execution time. This is a 'core' feature and does not depend on JUnit, TestNG or even Maven / Gradle.

Important: **do not** use the `@RunWith(Karate.class)` annotation. This is a *normal* JUnit test class !

```
import com.intuit.karate.cucumber.CucumberRunner;
import com.intuit.karate.cucumber.KarateStats;
import cucumber.api.CucumberOptions;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

@CucumberOptions(tags = {"~@ignore"})
public class TestParallel {

    @Test
    public void testParallel() {
        KarateStats stats = CucumberRunner.parallel(getClass(), 5,
"target/surefire-reports");
        assertTrue("scenarios failed", stats.getFailCount() == 0);
    }

}
```

Things to note:

- You don't use a JUnit runner (no `@RunWith` annotation), and you write a plain vanilla JUnit test (it could very well be TestNG or plain old Java) using the `CucumberRunner.parallel()` static method in `karate-core`.
- You can use the returned `KarateStats` to check if any scenarios failed.

- The first argument can be any class that marks the 'root package' in which `*.feature` files will be looked for, and sub-directories will be also scanned. As shown above you would typically refer to the enclosing test-class itself.
- The second argument is the number of threads to use.
- JUnit XML reports will be generated in the path you specify as the third parameter, and you can easily configure your CI to look for these files after a build (for e.g. in `**/*.xml` or `**/surefire-reports/*.xml`). This argument is optional and will default to `target/surefire-reports`.
- Cucumber JSON reports will be generated side-by-side with the JUnit XML reports and with the same name, except that the extension will be `.json` instead of `.xml`.
- No other reports will be generated. If you specify a `plugin` option via the `@CucumberOptions` annotation, or the command-line, or the 'maven-surefire-plugin' `<systemProperties>` - it will be ignored.
- But all other options passed to `@CucumberOptions` would work as expected, provided you point the `CucumberRunner` to the annotated class as the first argument. Note that in this example, any `*.feature` file tagged as `@ignore` will be skipped. You can also specify tags on the command-line.
- For convenience, some stats are logged to the console when execution completes, which should look something like this:

```
======================================================
elapsed time: 3.62 | total thread time: 13.79
features:    31 | threads:    5 | efficiency: 0.76
scenarios:   70 | failed:     0 | skipped:    0
======================================================
```

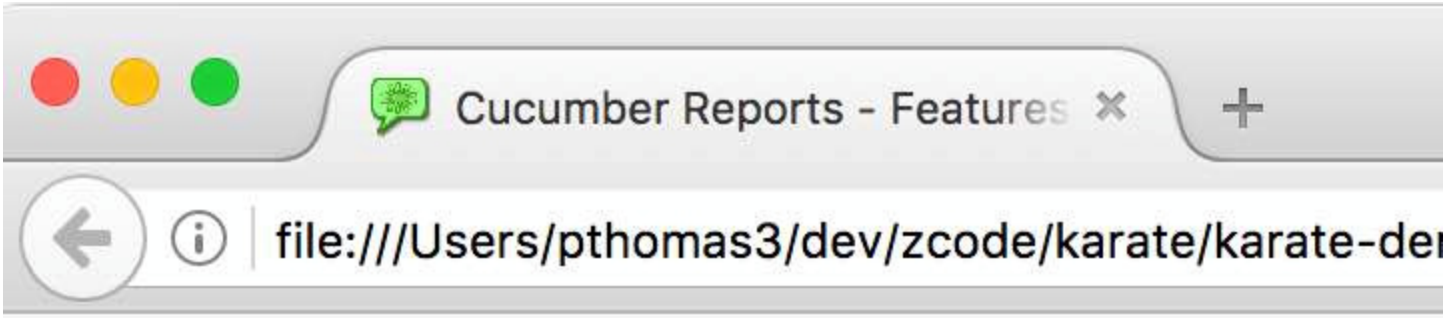
This is the preferred way of automating the execution of all Karate tests in a project, mainly because the other 'native' Cucumber reports (e.g. HTML) are not thread-safe.

Test Reports

As mentioned above, most CI tools would be able to process the JUnit XML output of the parallel runner and determine the status of the build as well as generate reports.

The Karate Demo has a working example of the recommended parallel-runner set up. It also details how a third-party library can be easily used to generate some very nice-looking reports, from the JSON output of the parallel runner.

For example, here below is an actual report generated by the cucumber-reporting open-source library.



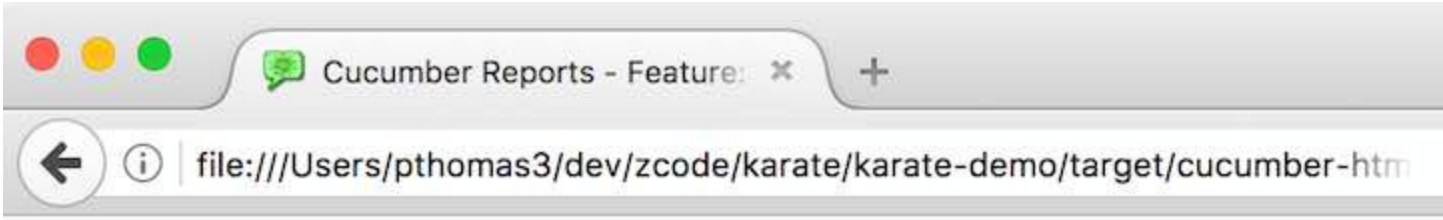
Cucumber Report

Features Statistics

The following graphs show passing and failing stati

Feature
demo/callarray/call-json-array.feature
demo/calldynamic/call-dynamic-json.feature

This report is recommended especially because Karate's integration includes the HTTP request and response logs in-line with the test report, which is extremely useful for troubleshooting test failures.



Cucumber Report

Feature Report

	Steps			
Feature	Passed	Failed	Skipped	Pendin
demo/callfeature/call-feature.feature	27	0	0	0

Feature demo/callfeature/call-feature.feature

calling another feature file

Background

Steps

* url demoBaseUrl

Scenario create kittens and then create parent cat

Steps

* call create-two-cats.feature
* url demoBaseUrl

Given path 'cats'

And request { name: 'Bob' }

When method post

Doc string

```
08:10:05.182 DEBUG -
1 > POST http://127.0.0.1:58317/cats
1 > Accept-Encoding: gzip,deflate
1 > Connection: Keep-Alive
1 > Content-Length: 14
1 > Content-Type: application/json
1 > Host: 127.0.0.1:58317
1 > User-Agent: Apache-HttpClient/4.5.3 (Java/1.8.0_112)
{"name":"Bob"}

08:10:05.418 DEBUG -
```

The demo also features code-coverage using Jacoco.

Logging

This is optional, and Karate will work without the logging config in place, but the default console logging may be too verbose for your needs.

Karate uses LOGBack which looks for a file called `logback-test.xml` on the 'classpath'.

Here is a sample `logback-test.xml` for you to get started.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>target/karate.log</file>
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <logger name="com.intuit.karate" level="DEBUG"/>

    <root level="info">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>

</configuration>
```

You can change the `com.intuit.karate` logger level to `INFO` to reduce the amount of logging. When the level is `DEBUG` the entire request and response payloads are logged.

Configuration

You can skip this section and jump straight to the Syntax Guide if you are in a hurry to get started with Karate. Things will work even if the `karate-config.js` file is not present.

Classpath

The 'classpath' is a Java concept and is where some configuration files such as the one for logging are expected to be by default. If you use the Maven `<test-resources>` tweak described earlier (recommended), the 'root' of the classpath will be in the `src/test/java` folder, or else would be `src/test/resources`.

karate-config.js

The only 'rule' is that on start-up Karate expects a file called `karate-config.js` to exist on the 'classpath' and contain a JavaScript function. The function is expected to return a JSON object and all keys and values in that JSON object will be made available as script variables.

And that's all there is to Karate configuration ! You can easily get the value of the current 'environment' or 'profile', and then set up 'global' variables using some simple JavaScript. Here is an example:

```
function() {
  var env = karate.env; // get java system property 'karate.env'
  karate.log('karate.env system property was:', env);
  if (!env) {
    env = 'dev'; // a custom 'intelligent' default
  }
  var config = { // base config JSON
    appId: 'my.app.id',
    appSecret: 'my.secret',
    someUrlBase: 'https://some-host.com/v1/auth/',
    anotherUrlBase: 'https://another-host.com/v1/'
  };
  if (env == 'stage') {
    // over-ride only those that need to be
    config.someUrlBase = 'https://stage-host/v1/auth';
  } else if (env == 'e2e') {
    config.someUrlBase = 'https://e2e-host/v1/auth';
  }
  // don't waste time waiting for a connection or if servers don't respond
within 5 seconds
  karate.configure('connectTimeout', 5000);
  karate.configure('readTimeout', 5000);
  return config;
}
```

The `karate` object has a few helper methods described in detail later in this document where the `call` keyword is explained. Here above, you see `karate.log()`, `karate.env` and `karate.configure()` being used. Note that the `karate-config.js` is re-invoked for *every* `Scenario` and in rare cases, you may want to initialize (e.g. auth tokens) only once for all of your tests. This can be achieved using `karate.callSingle()`.

A common requirement is to pass dynamic parameter values via the command line, and you can use the `karate.properties['some.name']` syntax for getting a system property passed via JVM options in the form `-Dsome.name=foo`. Refer to the section on dynamic port numbers for an example.

This decision to use JavaScript for config is influenced by years of experience with the set-up of complicated test-suites and fighting with Maven profiles, Maven resource-filtering and the XML-soup that somehow gets summoned by the Maven AntRun plugin.

Karate's approach frees you from Maven, is far more expressive, allows you to eyeball all environments in one place, and is still a plain-text file. If you want, you could even create nested chunks of JSON that 'name-space' your config variables.

One way to appreciate Karate's approach is to think over what it takes to add a new environment-dependent variable (e.g. a password) into a test. In typical frameworks it could mean changing multiple properties files, maven profiles and placeholders, and maybe even threading the value via a dependency-injection framework - before you can even access the value within your test.

This approach is indeed slightly more complicated than traditional `*.properties` files - but you *need* this complexity. Keep in mind that these are tests (not production code) and this config is going to be maintained more by the dev or QE team instead of the 'ops' or operations team.

And there is no more worrying about Maven profiles and whether the 'right' `*.properties` file has been copied to the proper place.

Switching the Environment

There is only one thing you need to do to switch the environment - which is to set a Java system property.

The recipe for doing this when running Maven from the command line is:

```
mvn test -DargLine="-Dkarate.env=e2e"
```

Or in Gradle:

```
./gradlew test -Dkarate.env=e2e
```

You can refer to the documentation of the Maven Surefire Plugin for alternate ways of achieving this, but the `argLine` approach is the simplest and should be more than sufficient for your Continuous Integration or test-automation needs.

Here's a reminder that running any single JUnit test via Maven can be done by:

```
mvn test -Dtest=CatsRunner
```

Where `CatsRunner` is the JUnit class name (in any package) you wish to run.

Karate is flexible, you can easily over-write config variables within each individual test-script - which is very convenient when in dev-mode or rapid-prototyping.

Just for illustrative purposes, you could 'hard-code' the `karate.env` for a specific JUnit test like this. Since CI test-automation would typically use a designated 'top-level suite' test-runner, you can actually have these individual test-runners lying around without any ill-effects. They are obviously useful for dev-mode troubleshooting. To ensure that they don't get run by CI by mistake - just *don't* use the `*Test.java` naming convention.

```
package animals.cats;

import com.intuit.karate.junit4.Karate;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;

@RunWith(Karate.class)
public class CatsRunner {

    @BeforeClass
    public static void before() {
        System.setProperty("karate.env", "pre-prod");
    }

}
```

Syntax Guide

Script Structure

Karate scripts are technically in 'Gherkin' format - but all you need to grok as someone who needs to test web-services are the three sections: `Feature`, `Background` and `Scenario`. There can be multiple Scenario-s in a `*.feature` file, and at least one should be present. The `Background` is optional.

Variables set using `def` in the `Background` will be re-set before *every* `Scenario`. If you are looking for a way to do something only **once** per `Feature`, take a look at `callonce`. On the other hand, if you are expecting a variable in the `Background` to be modified by one `Scenario` so that later ones can see the updated value - that is *not* how you should think of them, and you should combine your 'flow' into one scenario. Keep in mind that you should be able to comment-out a `Scenario` or skip some via `tags` without impacting any others.

Lines that start with a `#` are comments.

```
Feature: brief description of what is being tested
    more lines of description if needed.

Background:
# this section is optional !
# steps here are executed before each Scenario in this file
# variables defined here will be 'global' to all scenarios
# and will be re-initialized before every scenario
```

```
Scenario: brief description of this scenario
# steps for this scenario

Scenario: a different scenario
# steps for this other scenario
```

Given-When-Then

The business of web-services testing requires access to low-level aspects such as HTTP headers, URL-paths, query-parameters, complex JSON or XML payloads and response-codes. And Karate gives you control over these aspects with the small set of keywords focused on HTTP such as `url`, `path`, `param`, etc.

Karate does not attempt to have tests be in "natural language" like how Cucumber tests are traditionally expected to be. That said, the syntax is very concise, and the convention of every step having to start with either `Given`, `And`, `When` or `Then`, makes things very readable. You end up with a decent approximation of BDD even though web-services by nature are "headless", without a UI, and not really human-friendly.

Cucumber vs Karate

If you are familiar with Cucumber (JVM), you may be wondering if you need to write step-definitions. The answer is **no**.

Karate's approach is that all the step-definitions you need in order to work with HTTP, JSON and XML have been already implemented. And since you can easily extend Karate using JavaScript, there is no need to compile Java code any more.

The following table summmarizes some key differences between Cucumber and Karate.

	Cucumber	Karate
Step Definitions Built-In	**No**. You need to keep implementing them as your functionality grows. This can get very tedious, especially since for dependency-injection, you are on your own.	✔ **Yes**. No extra Java code needed.
Single Layer of Code To Maintain	**No**. There are 2 Layers. The Gherkin spec or `*.feature` files make up one layer, and you will also have the corresponding Java step-definitions.	✔ **Yes**. Only 1 layer of Karate-script (based on Gherkin).
Readable Specification	**Yes**. Cucumber will read like natural language *if* you implement the step-definitions right.	✘ **No**. Although Karate is simple, and a true DSL, it is ultimately a mini-programming language. But it is perfect for testing web-services at the level of HTTP requests and responses.

	Cucumber	Karate
Re-Use Feature Files	**No**. Cucumber does not support being able to call (and thus re-use) other `*.feature` files from a test-script.	✔ **Yes**.
Dynamic Data-Driven Testing	**No**. Cucumber's `Scenario Outline` expects the `Examples` to contain a fixed set of rows.	✔ **Yes**. Karate's support for calling other `*.feature` files allows you to use a JSON array as the data-source.
Parallel Execution	**No**. There are some challenges (especially with reporting) and you can find various discussions and third-party projects on the web that attempt to close this gap: 1 2 3 4 5 6 7 8	✔ **Yes**.
Run 'Set-Up' Routines Only Once	**No**. Cucumber has a limitation where `Background` steps are re-run for every `Scenario` and worse - even for every `Examples` row within a `Scenario Outline`. This has been a highly-requested open issue for a *long* time.	✔ **Yes**.

One nice thing about the design of the underlying Cucumber framework is that script-steps are treated the same no matter whether they start with the keyword `Given`, `And`, `When` or `Then`. What this means is that you are free to use whatever makes sense for you. You could even have all the steps start with `When` and Karate won't care.

In fact Cucumber supports the catch-all symbol '`*`' - instead of forcing you to use `Given`, `When` or `Then`. This is perfect for those cases where it really doesn't make sense - for example the `Background` section or when you use the `def` or `set` syntax. When eyeballing a test-script, think of the `*` as a 'bullet-point'.

You can read more about the Given-When-Then convention at the Cucumber reference documentation. Since Karate is based on Cucumber, you can also employ data-driven techniques such as expressing data-tables in test scripts. Another good thing that Karate inherits is the nice IDE support for Cucumber that IntelliJ and Eclipse have. So you can do things like right-click and run a `*.feature` file (or scenario) without needing to use a JUnit runner.

For a detailed discussion on BDD and how Karate relates to Cucumber, please refer to this blog-post: Yes, Karate is not *true* BDD. It is the opinion of the author of Karate that *true* BDD is un-necessary overkill for API testing, and this is explained more in this answer on Stack Overflow.

With the formalities out of the way, let's dive straight into the syntax.

Setting and Using Variables

`def`

Set a named variable

```
# assigning a string value:
Given def myVar = 'world'

# using a variable
Then print myVar

# assigning a number (you can use '*' instead of Given / When / Then)
* def myNum = 5
```

Note that `def` will over-write any variable that was using the same name earlier. Keep in mind that the start-up configuration routine could have already initialized some variables before the script even started.

The examples above are simple, but a variety of expression 'shapes' are supported on the right hand side of the = symbol. The section on Karate Expressions goes into the details.

`assert`

Assert if an expression evaluates to `true`

Once defined, you can refer to a variable by name. Expressions are evaluated using the embedded JavaScript engine. The assert keyword can be used to assert that an expression returns a boolean value.

```
Given def color = 'red '
And def num = 5
Then assert color + num == 'red 5'
```

Everything to the right of the `assert` keyword will be evaluated as a single expression.

Something worth mentioning here is that you would hardly need to use `assert` in your test scripts. Instead you would typically use the `match` keyword, that is designed for performing powerful assertions against JSON and XML response payloads.

`print`

Log to the console

You can use `print` to log variables to the console in the middle of a script. For convenience, you can have multiple expressions separated by commas, so this is the recommended pattern:

```
* print 'the value of a is:', a
```

Similar to `assert`, the expressions on the right-hand-side of a `print` have to be valid JavaScript. JsonPath and Karate expressions are not supported.

If you use commas (instead of concatenating strings using +), Karate will 'pretty-print' variables, which is what you typically want when dealing with JSON or XML.

```
* def myJson = { foo: 'bar', baz: [1, 2, 3] }
* print 'the value of myJson is:', myJson
```

Which results in the following output:

```
20:29:11.290 [main] INFO  com.intuit.karate - [print] the value of myJson is:
{
  "foo": "bar",
  "baz": [
    1,
    2,
    3
  ]
}
```

The built-in `karate` object is explained in detail later, but for now, note that this is also injected into `print` (and even `assert`) statements, and it has a helpful `pretty` method, that takes a JSON argument and a `prettyXml` method that deals with XML. So you could have also done something like:

```
* print 'the value of myJson is:\n' + karate.pretty(myJson)
```

Also refer to the `configure` keyword on how to switch on pretty-printing of all HTTP requests and responses.

'Native' data types

Native data types mean that you can insert them into a script without having to worry about enclosing them in strings and then having to 'escape' double-quotes all over the place. They seamlessly fit 'in-line' within your test script.

JSON

Note that the parser is 'lenient' so that you don't have to enclose all keys in double-quotes.

```
* def cat = { name: 'Billie', scores: [2, 5] }
* assert cat.scores[1] == 5
```

When asserting for expected values in JSON or XML you are probably better off using `match` instead of `assert`.

```
* def cats = [{ name: 'Billie' }, { name: 'Bob' }]
* match cats[1] == { name: 'Bob' }
```

Karate's native support for JSON means that you can assign parts of a JSON instance into another variable, which is useful when dealing with complex `response` payloads.

```
* def first = cats[0]
* match first == { name: 'Billie' }
```

For manipulating or updating JSON (or XML) using path expressions, refer to the `set` keyword.

XML

```
Given def cat =
<cat><name>Billie</name><scores><score>2</score><score>5</score></scores></ca
t>
# sadly, xpath list indexes start from 1
Then match cat/cat/scores/score[2] == '5'
# but karate allows you to traverse xml like json !!
Then match cat.cat.scores.score[1] == 5
```

Embedded Expressions

Karate has a very useful payload 'templating' approach. Variables can be referred to within JSON, for example:

```
Given def user = { name: 'john', age: 21 }
And def lang = 'en'
When def session = { name: '#(user.name)', locale: '#(lang)', sessionUser:
'#(user)'  }
```

So the rule is - if a string value within a JSON (or XML) object declaration is enclosed between `#(` and `)` - it will be evaluated as a JavaScript expression. And any variables which are alive in the context can be used in this expression. Here's how it works for XML:

```
Given def user = <user><name>john</name></user>
And def lang = 'en'
When def session =
<session><locale>#(lang)</locale><sessionUser>#(user)</sessionUser></session>
```

This comes in useful in some cases - and avoids needing to use the `set` keyword or JavaScript functions to manipulate JSON. So you get the best of both worlds: the elegance of JSON to express complex nested data - while at the same time being able to dynamically plug values (that could even be other JSON or XML 'trees') into a 'template'.

A few special built-in variables such as `$` (which is a reference to the JSON root) - can be mixed into JSON embedded expressions.

A special case of embedded expressions can remove a JSON key (or XML element / attribute) if the expression evaluates to `null`.

Enclosed JavaScript

An alternative to embedded expressions (for JSON only) is to enclose the entire payload within parentheses - which tells Karate to evaluate it as pure JavaScript. This can be a lot simpler than embedded expressions in many cases, and JavaScript programmers will feel right at home.

The example below shows the difference between embedded expressions and enclosed JavaScript:

```
When def user = { name: 'john', age: 21 }
And def lang = 'en'

* def embedded = { name: '#(user.name)', locale: '#(lang)', sessionUser:
'#(user)' }
* def enclosed = ({ name: user.name, locale: lang, sessionUser: user })
* match embedded == enclosed
```

So how would you choose between the two approaches to create JSON ? Embedded expressions are useful when you have complex JSON `read` from files, because you can auto-replace (or even remove) data-elements with values dynamically evaluated from variables. And the JSON will still be 'well-formed', and editable in your IDE or text-editor. Embedded expressions also make more sense in validation and schema-like short-cut situations. It can also be argued that the `#` symbol is easy to spot when eyeballing your test scripts - which makes things more readable and clear.

Multi-Line Expressions

The keywords `def`, `set`, `match`, `request` and `eval` take multi-line input as the last argument. This is useful when you want to express a one-off lengthy snippet of text in-line, without having to split it out into a separate file. Here are some examples:

```
# instead of:
* def cat =
<cat><name>Billie</name><scores><score>2</score><score>5</score></scores></ca
t>

# this is more readable:
* def cat =
"""
<cat>
    <name>Billie</name>
    <scores>
        <score>2</score>
        <score>5</score>
    </scores>
</cat>
"""
# example of a request payload in-line
Given request
"""
<?xml version='1.0' encoding='UTF-8'?>
```

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:QueryUsageBalance xmlns:ns2="http://www.mycompany.com/usage/V1">
    <ns2:UsageBalance>
        <ns2:LicenseId>12341234</ns2:LicenseId>
    </ns2:UsageBalance>
</ns2:QueryUsageBalance>
</S:Body>
</S:Envelope>
"""

# example of a payload assertion in-line
Then match response ==
"""
{ id: { domain: "DOM", type: "entityId", value: "#ignore" },
  created: { on: "#ignore" },
  lastUpdated: { on: "#ignore" },
  entityState: "ACTIVE"
}
"""
```

table

A simple way to create JSON Arrays

Now that we have seen how JSON is a 'native' data type that Karate understands, there is a very nice way to create JSON using Cucumber's support for expressing data-tables.

```
* table cats
    | name   | age |
    | 'Bob'  | 2   |
    | 'Wild' | 4   |
    | 'Nyan' | 3   |

* match cats == [{name: 'Bob', age: 2}, {name: 'Wild', age: 4}, {name:
'Nyan', age: 3}]
```

The `match` keyword is explained later, but it should be clear right away how convenient the `table` keyword is. JSON can be combined with the ability to call other `*.feature` files to achieve dynamic data-driven testing in Karate.

Notice that in the above example, string values within the table need to be enclosed in quotes. Otherwise they would be evaluated as expressions - which does come in useful for some dynamic data-driven situations:

```
* def one = 'hello'
* def two = { baz: 'world' }
* table json
    | foo     | bar            |
    | one     | { baz: 1 }     |
    | two.baz | ['baz', 'ban'] |
* match json == [{ foo: 'hello', bar: { baz: 1 } }, { foo: 'world', bar:
['baz', 'ban'] }]
```

Yes, you can even nest chunks of JSON in tables, and things work as you would expect.

Empty cells or expressions that evaluate to `null` will result in the key being omitted from the JSON. To force a `null` value, wrap it in parentheses:

```
* def one = { baz: null }
* table json
    | foo     | bar    |
    | 'hello' |        |
    | one.baz | (null) |
    | 'world' | null   |
* match json == [{ foo: 'hello' }, { bar: null }, { foo: 'world' }]
```

An alternate way to create data is using the `set` multiple syntax. It is actually a 'transpose' of the `table` approach, and can be very convenient when there are a large number of keys per row or if the nesting is complex. Here is an example of what is possible:

```
* set search
    | path       | 0        | 1      | 2        |
    | name.first | 'John'   | 'Jane' |          |
    | name.last  | 'Smith'  | 'Doe'  | 'Waldo'  |
    | age        | 20       |        |          |

* match search[0] == { name: { first: 'John', last: 'Smith' }, age: 20 }
* match search[1] == { name: { first: 'Jane', last: 'Doe' } }
* match search[2] == { name: { last: 'Waldo' } }
```

`text`

Don't parse, treat as raw text

Not something you would commonly use, but in some cases you need to disable Karate's default behavior of attempting to parse anything that looks like JSON (or XML) when using multi-line / string expressions. This is especially relevant when manipulating GraphQL queries - because although they look suspiciously like JSON, they are not, and tend to confuse Karate's internals. And as shown in the example below, having text 'in-line' is useful especially when you use the `Scenario Outline:` and `Examples:` for data-driven tests involving Cucumber-style place-holder substitutions in strings.

```
Scenario Outline:
# note the 'text' keyword instead of 'def'
* text query =
"""
{
  hero(name: "<name>") {
    height
    mass
  }
}
"""
Given path 'graphql'
```

```
And request { query: '#(query)' }
And header Accept = 'application/json'
When method post
Then status 200

Examples:
| name  |
| John  |
| Smith |
```

Note that if you did not need to inject `Examples:` into 'placeholders' enclosed within `<` and `>`, reading from a file with the extension `*.txt` may have been sufficient.

For placeholder-substitution, the `replace` keyword can be used instead, but with the advantage that the text can be read from a file or dynamically created.

Karate is a great fit for testing GraphQL because of how easy it is to deal with dynamic and deeply nested JSON responses. Refer to this example for more details: `graphql.feature`.

`replace`

Text Placeholder Replacement

Modifying existing JSON and XML is **natively** supported by Karate via the `set` keyword, and `replace` is primarily intended for dealing with raw strings. But when you deal with complex, nested JSON (or XML) - it may be easier in some cases to use `replace`, especially when you want to substitute multiple placeholders with one value, and when you don't need array manipulation. Since `replace` auto-converts the result to a string, make sure you perform type conversion back to JSON (or XML) if applicable.

Karate provides an elegant 'native-like' experience for placeholder substitution within strings or text content. This is useful in any situation where you need to concatenate dynamic string fragments to form content such as GraphQL or SQL.

The placeholder format defaults to angle-brackets, for example: `<replaceMe>`. Here is how to replace one placeholder at a time:

```
* def text = 'hello <foo> world'
* replace text.foo = 'bar'
* match text == 'hello bar world'
```

Karate makes it really easy to substitute multiple placeholders in a single, readable step as follows:

```
* def text = 'hello <one> world <two> bye'

* replace text
    | token | value   |
    | one   | 'cruel' |
```

```
    | two   | 'good'  |

* match text == 'hello cruel world good bye'
```

Note how strings have to be enclosed in quotes. This is so that you can mix expressions into text replacements as shown below. This example also shows how you can use a custom placeholder format instead of the default:

```
* def text = 'hello <one> world ${two} bye'
* def first = 'cruel'
* def json = { second: 'good' }

* replace text
    | token  | value       |
    | one    | first       |
    | ${two} | json.second |

* match text == 'hello cruel world good bye'
```

Refer to this file for a detailed example: `replace.feature`

yaml

Import YAML as JSON

For those who may prefer YAML as a simpler way to represent data, Karate allows you to read YAML content 'in-line' or even from a file - and it will be auto-converted to JSON.

```
# reading yaml 'in-line', note the 'yaml' keyword instead of 'def'
* yaml foo =
"""
name: John
input:
  id: 1
  subType:
    name: Smith
    deleted: false
"""
# the data is now JSON, so you can do JSON-things with it
* match foo ==
"""
{
  name: 'John',
  input: {
    id: 1,
    subType: { name: 'Smith', deleted: false }
  }
}
"""

# yaml from a file (the extension matters), and the data-type of 'bar' would
be JSON
* def bar = read('data.yaml')
```

JavaScript Functions

JavaScript Functions are also 'native'. And yes, functions can take arguments.
Standard JavaScript syntax rules apply.

ES6 arrow functions are **not** supported.

```
* def greeter = function(name){ return 'hello ' + name }
* assert greeter('Bob') == 'hello Bob'
```

When JavaScript executes in Karate, the built-in `karate` object provides some commonly used utility functions.

Java Interop

For more complex functions you are better off using the multi-line 'doc-string' approach. This example actually calls into existing Java code, and being able to do this opens up a whole lot of possibilities. The JavaScript interpreter will try to convert types across Java and JavaScript as smartly as possible. For e.g. JSON objects become Java `Map`-s, JSON arrays become Java `List`-s, and Java Bean properties are accessible (and update-able) using 'dot notation' e.g. '`object.name`'

```
* def dateStringToLong =
"""
function(s) {
  var SimpleDateFormat = Java.type('java.text.SimpleDateFormat');
  var sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
  return sdf.parse(s).time; // '.getTime()' would also have worked instead of
'.time'
}
"""
* assert dateStringToLong("2016-12-24T03:39:21.081+0000") == 1482550761081
```

More examples of Java interop and how to invoke custom code can be found in the section on Calling Java.

The `call` keyword provides an alternate way of calling JavaScript functions that have only one argument. The argument can be provided after the function name, without parentheses, which makes things slightly more readable (and less cluttered) especially when the solitary argument is JSON.

```
* def timeLong = call dateStringToLong '2016-12-24T03:39:21.081+0000'
* assert timeLong == 1482550761081

# a better example, with a JSON argument
* def greeter = function(name){ return 'Hello ' + name.first + ' ' +
name.last + '!' }
* def greeting = call greeter { first: 'John', last: 'Smith' }
```

Reading Files

Karate makes re-use of payload data, utility-functions and even other test-scripts as easy as possible. Teams typically define complicated JSON (or XML) payloads in a file and then re-use this in multiple scripts. Keywords such as `set` and `remove` allow you to to 'tweak' payload-data to fit the scenario under test. You can imagine how this greatly simplifies setting up tests for boundary conditions. And such re-use makes it easier to re-factor tests when needed, which is great for maintainability.

Note that the `set` (multiple) keyword can build complex, nested JSON (or XML) from scratch in a data-driven manner, and you may not even need to read from files for many situations. Test data can be within the main flow itself, which makes scripts highly readable.

Reading files is achieved using the `read` keyword. By default, the file is expected to be in the same folder (package) and side-by-side with the `*.feature` file. But you can prefix the name with `classpath:` in which case the 'root' folder would be `src/test/java` (assuming you are using the recommended folder structure).

Prefer `classpath:` when a file is expected to be heavily re-used all across your project. And yes, relative paths will work.

```
# json
* def someJson = read('some-json.json')
* def moreJson = read('classpath:more-json.json')

# xml
* def someXml = read('../common/my-xml.xml')

# import yaml (will be converted to json)
* def jsonFromYaml = read('some-data.yaml')

# string
* def someString = read('classpath:messages.txt')

# javascript (will be evaluated)
* def someValue = read('some-js-code.js')

# if the js file evaluates to a function, it can be re-used later using the
'call' keyword
* def someFunction = read('classpath:some-reusable-code.js')
* def someCallResult = call someFunction

# the following short-cut is also allowed
* def someCallResult = call read('some-js-code.js')
```

You can also re-use other `*.feature` files from test-scripts:

```
# perfect for all those common authentication or 'set up' flows
* def result = call read('classpath:some-reusable-steps.feature')
```

If a file does not end in `.json`, `.xml`, `.yaml`, `.js` or `.txt` - it is treated as a stream which is typically what you would need for `multipart` file uploads.

```
* def someStream = read('some-pdf.pdf')
```

The `.graphql` and `.gql` extensions are also recognized (for GraphQL) but are handled the same way as `.txt` and treated as a string.

For JSON and XML files, Karate will evaluate any embedded expressions on load. This enables more concise tests, and the file can be re-usable in multiple, data-driven tests.

Since it is internally implemented as a JavaScript function, you can mix calls to `read()` freely wherever JavaScript expressions are allowed:

```
* def someBigString = read('first.txt') + read('second.txt')
```

Tip: you can even use JS expressions to dynamically choose a file based on some condition: `* def someConfig = read('my-config-' + someVariable + '.json')`. Refer to conditional logic for more ideas.

And a very common need would be to use a file as the `request` body:

```
Given request read('some-big-payload.json')
```

Or in a `match`:

```
And match response == read('expected-response-payload.json')
```

The rarely used `file:` prefix is also supported. You could use it for 'hard-coded' absolute paths in dev mode, but is obviously not recommended for CI test-suites. A good example of where you may need this is if you programmatically write a file to the `target` folder, and then you can read it ike this:

```
* def payload = read('file:target/large.xml')
```

Take a look at the Karate Demos for real-life examples of how you can use files for validating HTTP responses, like this one: `read-files.feature`.

Type Conversion

Internally, Karate will auto-convert JSON (and even XML) to Java `Map` objects. And JSON arrays would become Java `List`-s. But you will never need to worry about this internal data-representation most of the time.

In some rare cases, for e.g. if you acquired a string from some external source, or if you generated JSON (or XML) by concatenating text or using `replace`, you may want to convert a

string to JSON and vice-versa. You can even perform a conversion from XML to JSON if you want.

One example of when you may want to convert JSON (or XML) to a string is when you are passing a payload to custom code via Java interop. Do note that when passing JSON, the default `Map` and `List` representations should suffice for most needs (see example), and using them would avoid un-necessary string-conversion.

So you have the following type markers you can use instead of `def` (or the rarely used `text`):

- `string` - convert JSON or any other data-type (except XML) to a string
- `json` - convert XML, a map-like or list-like object, a string, or even a Java bean (POJO) into JSON
- `xml` - convert JSON, a map-like object, a string, or even a Java bean (POJO) into XML
- `xmlstring` - specifically for converting the map-like Karate internal representation of XML into a string
- `copy` - to clone a given payload variable reference (JSON, XML, Map or List), refer: `copy`

These are best explained in this example file: `type-conv.feature`

If you want to 'pretty print' a JSON or XML value with indenting, refer to the documentation of the `print` keyword.

Floats and Integers

While converting a number to a string is easy (just concatenate an empty string e.g. `myInt + ''`), in some rare cases, you may need to convert a string to a number. You can do this by multiplying by `1` or using the built-in JavaScript `parseInt()` function:

```
* def foo = '10'
* string json = { bar: '#(1 * foo)' }
* match json == '{"bar":10.0}'

* string json = { bar: '#(parseInt(foo))' }
* match json == '{"bar":10.0}'
```

As per the JSON spec, all numeric values are treated as doubles, so for integers - it really doesn't matter if there is a decimal point or not. In fact it may be a good idea to slip doubles instead of integers into some of your tests ! Anyway, there are times when you may want to force integers (perhaps for cosmetic reasons) and you can easily do so using the 'double-tilde' short-cut: '~~'.

```
* def foo = '10'
* string json = { bar: '#(~~foo)' }
* match json == '{"bar":10}'

# unfortunately JS math always results in a double
* def foo = 10
```

```
* string json = { bar: '#(1 * foo)' }
* match json == '{"bar":10.0}'

# but you can easily coerce to an integer if needed
* string json = { bar: '#(~~(1 * foo))' }
* match json == '{"bar":10}'
```

Karate Expressions

Before we get to the HTTP keywords, it is worth doing a recap of the various 'shapes' that the right-hand-side of an assignment statement can take:

Example	Shape	Description
`* def foo = 'bar'`	JS	simple strings, numbers or booleans
`* def foo = 'bar' + baz[0]`	JS	any valid JavaScript expression, and variables can be mixed in, another example: `bar.length + 1`
`* def foo = { bar: '#(baz)' }`	JSON	anything that starts with a `{` or a `[` is parsed as JSON, use `text` instead of `def` if you need to suppress the default behavior
`* def foo = ({ bar: baz })`	JS	enclosed JavaScript, the result of which is exactly equivalent to the above
`* def foo = <foo>bar</foo>`	XML	anything that starts with a `<` is parsed as XML, use `text` instead of `def` if you need to suppress the default behavior
`* def foo = function(arg){ return arg + bar }`	JS Fn	anything that starts with `function(...){` is parsed as a JS function.
`* def foo = read('bar.json')`	JS	using the built-in `read()` function
`* def foo = $.bar[0]`	JsonPath	short-cut JsonPath on the `response`
`* def foo = /bar/baz`	XPath	short-cut XPath on the `response`
`* def foo = get bar $..baz[?(@.ban)]`	`get` JsonPath	JsonPath on the variable `bar`, you can also use `get[0]` to get the first item if the JsonPath evaluates to an array - especially useful when using wildcards such as `[*]` or filter-criteria
`* def foo = $bar..baz[?(@.ban)]`	$var.JsonPath	convenience short-cut for the above
`* def foo = get bar count(/baz//ban)`	`get` XPath	XPath on the variable `bar`
`* def foo = karate.pretty(bar)`	JS	using the built-in `karate` object in JS expressions
`* def Foo = Java.type('com.mycompany.Foo')`	JS-Java	Java Interop, and even package-name-spaced one-liners like `java.lang.System.currentTimeMillis(`

Example	Shape	Description
		`)` are possible
`* def foo = call bar { baz: '#(ban)' }`	`call`	or `callonce`, where expressions like `read('foo.js')` are allowed as the object to be called or the argument
`* def foo = bar({ baz: ban })`	JS	equivalent to the above, JavaScript function invocation

Core Keywords

They are `url`, `path`, `request`, `method` and `status`.

These are essential HTTP operations, they focus on setting one (un-named or 'key-less') value at a time and therefore don't need an = sign in the syntax.

`url`

```
Given url 'https://myhost.com/v1/cats'
```

A URL remains constant until you use the `url` keyword again, so this is a good place to set-up the 'non-changing' parts of your REST URL-s.

A URL can take expressions, so the approach below is legal. And yes, variables can come from global config.

```
Given url 'https://' + e2eHostName + '/v1/api'
```

If you are trying to build dynamic URLs including query-string parameters in the form: `http://myhost/some/path?foo=bar&search=true` - please refer to the `param` keyword.

`path`

REST-style path parameters. Can be expressions that will be evaluated. Comma delimited values are supported which can be more convenient, and takes care of URL-encoding and appending '/' where needed.

```
Given path 'documents/' + documentId + '/download'

# this is equivalent to the above
Given path 'documents', documentId, 'download'

# or you can do the same on multiple lines if you wish
Given path 'documents'
And path documentId
And path 'download'
```

Note that the `path` 'resets' after any HTTP request is made but not the `url`. The Hello World is a great example of 'REST-ful' use of the `url` when the test focuses on a single REST 'resource'. Look at how the `path` did not need to be specified for the second HTTP `get` call since `/cats` is part of the `url`.

Important: If you attempt to build a URL in the form `?myparam=value` by using `path` the `?` will get encoded into `%3F`. Use either the `param` keyword, e.g.: `* param myparam = 'value'` or `url`: `* url 'http://example.com/v1?myparam'`

request

In-line JSON:

```
Given request { name: 'Billie', type: 'LOL' }
```

In-line XML:

```
And request <cat><name>Billie</name><type>Ceiling</type></cat>
```

From a file in the same package. Use the `classpath:` prefix to load from the classpath instead.

```
Given request read('my-json.json')
```

You could always use a variable:

```
And request myVariable
```

In most cases you won't need to set the `Content-Type header` as Karate will automatically do the right thing depending on the data-type of the `request`.

Defining the `request` is mandatory if you are using an HTTP `method` that expects a body such as `post`. If you really need to have an empty body, you can use an empty string as shown below, and you can force the right `Content-Type` header by using the `header` keyword.

```
Given request ''
And header Content-Type = 'text/html'
```

Sending a file as the entire binary request body is easy (note that `multipart` is different):

```
Given path 'upload'
And request read('my-image.jpg')
When method put
Then status 200
```

method

The HTTP verb - `get`, `post`, `put`, `delete`, `patch`, `options`, `head`, `connect`, `trace`.

Lower-case is fine.

```
When method post
```

It is worth internalizing that during test-execution, it is upon the `method` keyword that the actual HTTP request is issued. Which suggests that the step should be in the `When` form, for example: `When method post`. And steps that follow should logically be in the `Then` form. Also make sure that you complete the set up of things like `url`, `param`, `header`, `configure` etc. *before* you fire the `method`.

```
# set headers or params (if any) BEFORE the method step
Given header Accept = 'application/json'
When method get
# the step that immediately follows the above would typically be:
Then status 200
```

Although rarely needed, variable references or expressions are also supported:

```
* def putOrPost = (someVariable == 'dev' ? 'put' : 'post')
* method putOrPost
```

`status`

This is a shortcut to assert the HTTP response code.

```
Then status 200
```

And this assertion will cause the test to fail if the HTTP response code is something else.

See also `responseStatus`.

Keywords that set key-value pairs

They are `param`, `header`, `cookie`, `form field` and `multipart field`.

The syntax will include a '=' sign between the key and the value. The key should not be within quotes.

To make dynamic data-driven testing easier, the following keywords also exist: `params`, `headers`, `cookies` and `form fields`. They use JSON to build the relevant parts of the HTTP request.

`param`

Setting query-string parameters:

```
Given param someKey = 'hello'
```

```
And param anotherKey = someVariable
```

The above would result in a URL like:
`http://myhost/mypath?someKey=hello&anotherKey=foo`. Note that the `?` and `&` will be automatically inserted.

Multi-value params are also supported:

```
* param myParam = 'foo', 'bar'
```

You can also use JSON to set multiple query-parameters in one-line using `params` and this is especially useful for dynamic data-driven testing.

`header`

You can use functions or expressions:

```
Given header Authorization = myAuthFunction()
And header transaction-id = 'test-' + myIdString
```

It is worth repeating that in most cases you won't need to set the `Content-Type` header as Karate will automatically do the right thing depending on the data-type of the `request`.

Because of how easy it is to set HTTP headers, Karate does not provide any special keywords for things like the `Accept` header. You simply do something like this:

```
Given path 'some/path'
And request { some: 'data' }
And header Accept = 'application/json'
When method post
Then status 200
```

A common need is to send the same header(s) for *every* request, and `configure headers` (with JSON) is how you can set this up once for all subsequent requests. And if you do this within a `Background:` section, it would apply to all `Scenario:` sections within the `*.feature` file.

```
* configure headers = { 'Content-Type': 'application/xml' }
```

Note: in this example above, `Content-Type` had to be enclosed in quotes because the hyphen `-` would cause problems otherwise.

If you need headers to be dynamically generated for each HTTP request, use a JavaScript function with `configure headers` instead of JSON.

Multi-value headers (though rarely used in the wild) are also supported:

```
* header myHeader = 'foo', 'bar'
```

Also look at the `headers` keyword which uses JSON and makes some kinds of dynamic data-driven testing easier.

`cookie`

Setting a cookie:

```
Given cookie foo = 'bar'
```

You also have the option of setting multiple cookies in one-step using the `cookies` keyword.

Note that any cookies returned in the HTTP response would be automatically set for any future requests. This mechanism works by calling `configure cookies` behind the scenes and if you need to stop auto-adding cookies for future requests, just do this:

```
* configure cookies = null
```

Also refer to the built-in variable `responseCookies` for how you can access and perform assertions on cookie data values.

`form field`

HTML form fields would be URL-encoded when the HTTP request is submitted (by the `method` step). You would typically use these to simulate a user sign-in and then grab a security token from the `response`. For example:

```
Given path 'login'
And form field username = 'john'
And form field password = 'secret'
When method post
Then status 200
And def authToken = response.token
```

A good example of the use of `form field` for a typical sign-in flow is this OAuth 2 demo: `oauth2.feature`.

Multi-values are supported the way you would expect (e.g. for simulating check-boxes and multi-selects):

```
* form field selected = 'apple', 'orange'
```

You can also dynamically set multiple fields in one step using the `form fields` keyword.

`multipart field`

Use this for building multipart named (form) field requests. This is typically combined with `multipart file` as shown below.

multipart file

```
Given multipart file myFile = { read: 'test.pdf', filename: 'upload-
name.pdf', contentType: 'application/pdf' }
And multipart field message = 'hello world'
When method post
Then status 200
```

Note that `multipart file` takes a JSON argument so that you can easily set the `filename` and the `contentType` (mime-type) in one step.

- `read`: mandatory, - the name of a file, and the `classpath:` prefix also is allowed.
- `filename`: optional, will default to the multipart field name if not specified
- `contentType`: optional, will default to `application/octet-stream`

When 'multipart' content is involved, the `Content-Type` header of the HTTP request defaults to `multipart/form-data`. You can over-ride it by using the `header` keyword before the `method` step. Look at `multipart entity` for an example.

Also refer to this demo example for a working example of multipart file uploads: `upload.feature`.

You can also dynamically set multiple files in one step using `multipart files`.

multipart entity

This is technically not in the key-value form: `multipart field name = 'foo'`, but logically belongs here in the documentation.

Use this for multipart content items that don't have field-names. Here below is an example that also demonstrates using the `multipart/related` content-type.

```
Given path '/v2/documents'
And multipart entity read('foo.json')
And multipart field image = read('bar.jpg')
And header Content-Type = 'multipart/related'
When method post
Then status 201
```

Multi-Param Keywords

Keywords that set multiple key-value pairs in one step

`params`, `headers`, `cookies`, `form fields` and `multipart files` take a single JSON argument (which can be in-line or a variable reference), and this enables certain types of dynamic data-driven testing, especially because any JSON key with a `null` value will be ignored. Here is a good example in the demos: `dynamic-params.feature`

params

```
* params { searchBy: 'client', active: true, someList: [1, 2, 3] }
```

See also `param`.

headers

```
* def someData = { Authorization: 'sometoken', tx_id: '1234', extraTokens:
['abc', 'def'] }
* headers someData
```

See also `header`.

cookies

```
* cookies { someKey: 'someValue', foo: 'bar' }
```

See also `cookie`.

form fields

```
* def credentials = { username: '#(user.name)', password: 'secret', projects:
['one', 'two'] }
* form fields credentials
```

See also `form field`.

multipart files

The single JSON argument needs to be in the form `{ field1: { read: 'file1.ext' }, field2: { read: 'file2.ext' } }` where each nested JSON is in the form expected by `multipart file`

```
* def json = {}
* set json.myFile1 = { read: 'test1.pdf', filename: 'upload-name1.pdf',
contentType: 'application/pdf' }
# if you have dynamic keys you can do this
* def key = 'myFile2'
* eval json[key] = { read: 'test2.pdf', filename: 'upload-name2.pdf',
contentType: 'application/pdf' }
And multipart files json
```

SOAP

Since a SOAP request needs special handling, this is the only case where the `method` step is not used to actually fire the request to the server.

soap action

The name of the SOAP action specified is used as the 'SOAPAction' header. Here is an example which also demonstrates how you could assert for expected values in the response XML.

```
Given request read('soap-request.xml')
When soap action 'QueryUsageBalance'
Then status 200
And match response /Envelope/Body/QueryUsageBalanceResponse/Result/Error/Code
== 'DAT_USAGE_1003'
And match response /Envelope/Body/QueryUsageBalanceResponse ==
read('expected-response.xml')
```

A working example of calling a SOAP service can be found within the Karate project test-suite. Refer to the demos for another example: `soap.feature`.

More examples are available that showcase various ways of parameter-izing and dynamically manipulating SOAP requests in a data-driven fashion. Karate is quite flexible, and provides multiple options for you to evolve patterns that fit your environment, as you can see here: `xml.feature`.

`configure`

Managing Headers, SSL, Timeouts and HTTP Proxy

You can adjust configuration settings for the HTTP client used by Karate using this keyword. The syntax is similar to `def` but instead of a named variable, you update configuration. Here are the configuration keys supported:

Key	Type	Description
`headers`	JSON / JS function	See `configure headers`
`cookies`	JSON / JS function	Just like `configure headers`, but for cookies. You will typically never use this, as response cookies are auto-added to all future requests. If you need to clear cookies at any time, just do `configure cookies = null`
`logPrettyRequest`	boolean	Pretty print the request payload JSON or XML with indenting (default `false`)
`logPrettyResponse`	boolean	Pretty print the response payload JSON or XML with indenting (default `false`)
`printEnabled`	boolean	Can be used to suppress the `print` output when not in 'dev mode' by setting as `false` (default `true`)
`afterScenario`	JS function	Will be called after every `Scenario` (or `Example` within a `Scenario Outline`), refer to this example: `hooks.feature`
`afterFeature`	JS function	Will be called after every `Feature`, refer to this example: `hooks.feature`
`ssl`	boolean	Enable HTTPS calls without needing to configure a trusted

Key	Type	Description
		certificate or key-store.
`ssl`	string	Like above, but force the SSL algorithm to one of these values. (The above form internally defaults to `TLS` if simply set to `true`).
`ssl`	JSON	see X509 certificate authentication
`followRedirects`	boolean	Whether the HTTP client automatically follows redirects - (default `true`), refer to this example.
`connectTimeout`	integer	Set the connect timeout (milliseconds). The default is 30000 (30 seconds).
`readTimeout`	integer	Set the read timeout (milliseconds). The default is 30000 (30 seconds).
`proxy`	string	Set the URI of the HTTP proxy to use.
`proxy`	JSON	For a proxy that requires authentication, set the `uri`, `username` and `password`. (See example below).
`charset`	string	The charset that will be sent in the request `Content-Type` which defaults to `utf-8`. You typically never need to change this, and you can over-ride this per-request if needed via the `header` keyword (example).
`httpClientClass`	string	See karate-mock-servlet
`httpClientInstance`	Java Object	See karate-mock-servlet
`userDefined`	JSON	See karate-mock-servlet
`responseHeaders`	JSON / JS function	See karate-netty
`cors`	boolean	See karate-netty

Examples:

```
# pretty print the response payload
* configure logPrettyResponse = true

# enable ssl (and no certificate is required)
* configure ssl = true

# enable ssl and force the algorithm to TLSv1.2
* configure ssl = 'TLSv1.2'

# time-out if the response is not received within 10 seconds (after the
connection is established)
* configure readTimeout = 10000

# set the uri of the http proxy server to use
* configure proxy = 'http://my.proxy.host:8080'

# proxy which needs authentication
```

```
* configure proxy = { uri: 'http://my.proxy.host:8080', username: 'john',
password: 'secret' }
```

And if you need to set some of these 'globally' you can easily do so using the `karate` object in `karate-config.js`.

System Properties for SSL and HTTP proxy

For HTTPS / SSL, you can also specify a custom certificate or trust store by setting Java system properties. And similarly - for specifying the HTTP proxy.

X509 Certificate Authentication

Also referred to as "mutual auth" - if your API requires that clients present an X509 certificate for authentication, Karate supports this via JSON as the `configure ssl` value. The following parameters are supported:

Key	Type	Required?	Description
`keyStore`	string	required	path to file containing public and private keys for your client certificate.
`keyStorePassword`	string	required	password for keyStore file.
`keyStoreType`	string	required	Format of the keyStore file. Allowed keystore types are as described in the Java `KeyStore` docs.
`trustStore`	string	optional	path to file containing the trust chain for your server certificate.
`trustStorePassword`	string	optional	password for trustStore file.
`trustStoreType`	string	optional	Format of the trustStore file. Allowed keystore types are as described in the Java `KeyStore` docs.
`trustAll`	boolean	optional	if all server certificates should be considered trusted. Default is `true` and if the above 3 keys are present will allow self-signed certificates. If `false`, will expect the whole chain in the `trustStore` or use what is available in the environment.
`algorithm`	string	optional	force the SSL algorithm to one of these values. Default is `TLS`.

Example:

```
# enable X509 certificate authentication with PKCS12 file 'certstore.pfx' and
password 'certpassword'
* configure ssl = { keyStore: 'classpath:certstore.pfx', keyStorePassword:
'certpassword', keyStoreType: 'pkcs12' };
```

Payload Assertions

Prepare, Mutate, Assert.

Now it should be clear how Karate makes it easy to express JSON or XML. If you read from a file, the advantage is that multiple scripts can re-use the same data.

Once you have a JSON or XML object, Karate provides multiple ways to manipulate, extract or transform data. And you can easily assert that the data is as expected by comparing it with another JSON or XML object.

`match`

Payload Assertions / Smart Comparison

The `match` operation is smart because white-space does not matter, and the order of keys (or data elements) does not matter. Karate is even able to ignore fields you choose - which is very useful when you want to handle server-side dynamically generated fields such as UUID-s, time-stamps, security-tokens and the like.

The match syntax involves a double-equals sign '==' to represent a comparison (and not an assignment '=').

Since `match` and `set` go well together, they are both introduced in the examples in the section below.

`set`

Manipulating Data

Game, `set` and `match` - Karate !

Setting values on JSON documents is simple using the `set` keyword and JsonPath expressions.

```
* def myJson = { foo: 'bar' }
* set myJson.foo = 'world'
* match myJson == { foo: 'world' }

# add new keys.  you can use pure JsonPath expressions (notice how this is
different from the above)
* set myJson $.hey = 'ho'
* match myJson == { foo: 'world', hey: 'ho' }

# and even append to json arrays (or create them automatically)
* set myJson.zee[0] = 5
* match myJson == { foo: 'world', hey: 'ho', zee: [5] }

# omit the array index to append
* set myJson.zee[] = 6
* match myJson == { foo: 'world', hey: 'ho', zee: [5, 6] }
```

```
# nested json ? no problem
* set myJson.cat = { name: 'Billie' }
* match myJson == { foo: 'world', hey: 'ho', zee: [5, 6], cat: { name:
'Billie' } }

# and for match - the order of keys does not matter
* match myJson == { cat: { name: 'Billie' }, hey: 'ho', foo: 'world', zee:
[5, 6] }

# you can ignore fields marked with '#ignore'
* match myJson == { cat: '#ignore', hey: 'ho', foo: 'world', zee: [5, 6] }
```

XML and XPath works just like you'd expect.

```
* def cat = <cat><name>Billie</name></cat>
* set cat /cat/name = 'Jean'
* match cat / == <cat><name>Jean</name></cat>

# you can even set whole fragments of xml
* def xml = <foo><bar>baz</bar></foo>
* set xml/foo/bar = <hello>world</hello>
* match xml == <foo><bar><hello>world</hello></bar></foo>
```

Refer to the section on XPath Functions for examples of advanced XPath usage.

`match` and variables

In case you were wondering, variables (and even expressions) are supported on the right-hand-side. So you can compare 2 JSON (or XML) payloads if you wanted to:

```
* def foo = { hello: 'world', baz: 'ban' }
* def bar = { baz: 'ban', hello: 'world' }
* match foo == bar
```

If you are wondering about the finer details of the `match` syntax, the left-hand-side has to be either a variable name, or a 'named' JsonPath or XPath expression. And the right-hand-side can be any valid Karate expression.

Refer to the section on JsonPath short-cuts for a deeper understanding of 'named' JsonPath expressions in Karate.

`match !=` (not equals)

The 'not equals' operator `!=` works as you would expect:

```
* def test = { foo: 'bar' }
* match test != { foo: 'baz' }
```

You typically will *never* need to use the `!=` (not-equals) operator ! Use it sparingly, and only for string, number or simple payload comparisons.

`set` multiple

Karate has an elegant way to set multiple keys (via path expressions) in one step. For convenience, non-existent keys (or array elements) will be created automatically. You can find more JSON examples here: `js-arrays.feature`.

```
* def cat = { name: '' }

* set cat
| path   | value |
| name   | 'Bob' |
| age    | 5     |

* match cat == { name: 'Bob', age: 5 }
```

One extra convenience for JSON is that if the variable itself (which was `cat` in the above example) does not exist, it will be created automatically. You can even create (or modify existing) JSON arrays by using multiple columns.

```
* set foo
| path | 0     | 1     |
| bar  | 'baz' | 'ban' |

* match foo == [{ bar: 'baz' }, { bar: 'ban' }]
```

If you have to set a bunch of deeply nested keys, you can move the parent path to the top, next to the `set` keyword and save a lot of typing !

```
* set foo.bar
| path   | value |
| one    | 1     |
| two[0] | 2     |
| two[1] | 3     |

* match foo == { bar: { one: 1, two: [2, 3] } }
```

The same concept applies to XML and you can build complicated payloads from scratch in just a few, extremely readable lines. The `value` column can take expressions, *even* XML chunks. You can find more examples here: `xml.feature`.

```
* set search /acc:getAccountByPhoneNumber
| path                         | value |
| acc:phone/@foo               | 'bar' |
| acc:phone/acc:number[1]      | 1234  |
| acc:phone/acc:number[2]      | 5678  |
| acc:phoneNumberSearchOption | 'all' |

* match search ==
"""
<acc:getAccountByPhoneNumber>
    <acc:phone foo="bar">
        <acc:number>1234</acc:number>
```

```
        <acc:number>5678</acc:number>
    </acc:phone>
    <acc:phoneNumberSearchOption>all</acc:phoneNumberSearchOption>
</acc:getAccountByPhoneNumber>
"""
```

`remove`

This is like the opposite of `set` if you need to remove keys or data elements from JSON or XML instances. You can even remove JSON array elements by index.

```
* def json = { foo: 'world', hey: 'ho', zee: [1, 2, 3] }
* remove json.hey
* match json == { foo: 'world', zee: [1, 2, 3] }
* remove json $.zee[1]
* match json == { foo: 'world', zee: [1, 3] }
```

`remove` works for XML elements as well:

```
* def xml = <foo><bar><hello>world</hello></bar></foo>
* remove xml/foo/bar/hello
* match xml == <foo><bar/></foo>
* remove xml /foo/bar
* match xml == <foo/>
```

Also take a look at how a special case of embedded-expressions can remove key-value pairs from a JSON (or XML) payload: Remove if Null.

Fuzzy Matching

Ignore or Validate

When expressing expected results (in JSON or XML) you can mark some fields to be ignored when the match (comparison) is performed. You can even use a regular-expression so that instead of checking for equality, Karate will just validate that the actual value conforms to the expected pattern.

This means that even when you have dynamic server-side generated values such as UUID-s and time-stamps appearing in the response, you can still assert that the full-payload matched in one step.

```
* def cat = { name: 'Billie', type: 'LOL', id: 'a9f7a56b-8d5c-455c-9d13-
808461d17b91' }
* match cat == { name: '#ignore', type: '#regex [A-Z]{3}', id: '#uuid' }
# this will fail
# * match cat == { name: '#ignore', type: '#regex .{2}', id: '#uuid' }
```

The supported markers are the following:

Marker	Description
`#ignore`	Skip comparison for this field even if the data element or JSON key is present
`#null`	Expects actual value to be `null`, and the data element or JSON key *must* be present
`#notnull`	Expects actual value to be not-`null`
`#present`	Actual value can be any type or *even* `null`, and the element or JSON key *must* be present
`#notpresent`	Expects the data element or JSON key to be **not** present at all
`#array`	Expects actual value to be a JSON array
`#object`	Expects actual value to be a JSON object
`#boolean`	Expects actual value to be a boolean `true` or `false`
`#number`	Expects actual value to be a number
`#string`	Expects actual value to be a string
`#uuid`	Expects actual (string) value to conform to the UUID format
`#regex STR`	Expects actual (string) value to match the regular-expression 'STR' (see examples above)
`#? EXPR`	Expects the JavaScript expression 'EXPR' to evaluate to true, see self-validation expressions below
`#[NUM] EXPR`	Advanced array validation, see schema validation
`#(EXPR)`	For completeness, embedded expressions belong in this list as well

Optional Fields

If two cross-hatch `#` symbols are used as the prefix (for example: `##number`), it means that the key is optional or that the value can be null.

```
* def foo = { bar: 'baz' }
* match foo == { bar: '#string', ban: '##string' }
```

Remove If Null

A very useful behavior when you combine the optional marker with an embedded expression is as follows: if the embedded expression evaluates to `null` - the JSON key (or XML element or attribute) will be deleted from the payload (the equivalent of `remove`).

```
* def data = { a: 'hello', b: null, c: null }
* def json = { foo: '#(data.a)', bar: '#(data.b)', baz: '##(data.c)' }
* match json == { foo: 'hello', bar: null }
```

`#null` and `#notpresent`

Karate's `match` is strict, and the case where a JSON key exists but has a `null` value (`#null`) is considered different from the case where the key is not present at all (`#notpresent`) in the payload.

But note that `##null` can be used to represent a convention that many teams adopt, which is that keys with `null` values are stripped from the JSON payload. In other words, `{ a: 1, b: null }` is considered 'equal' to `{ a: 1 }` and `{ a: 1, b: '##null' }` will `match` both cases.

These examples (all exact matches) can make things more clear:

```
* def foo = { }
* match foo == { a: '##null' }
* match foo == { a: '##notnull' }
* match foo == { a: '#notpresent' }
* match foo == { a: '#ignore' }

* def foo = { a: null }
* match foo == { a: '#null' }
* match foo == { a: '##null' }
* match foo == { a: '#present' }
* match foo == { a: '#ignore' }

* def foo = { a: 1 }
* match foo == { a: '#notnull' }
* match foo == { a: '##notnull' }
* match foo == { a: '#present' }
* match foo == { a: '#ignore' }
```

'Self' Validation Expressions

The special 'predicate' marker `#? EXPR` in the table above is an interesting one. It is best explained via examples.

Observe how the value of the field being validated (or 'self') is injected into the 'underscore' expression variable: '_'

```
* def date = { month: 3 }
* match date == { month: '#? _ > 0 && _ < 13' }
```

What is even more interesting is that expressions can refer to variables:

```
* def date = { month: 3 }
* def min = 1
* def max = 12
* match date == { month: '#? _ >= min && _ <= max' }
```

And functions work as well ! You can imagine how you could evolve a nice set of utilities that validate all your domain objects.

```
* def date = { month: 3 }
```

```
* def isValidMonth = function(m) { return m >= 0 && m <= 12 }
* match date == { month: '#? isValidMonth(_)' }
```

Especially since strings can be easily coerced to numbers (and vice-versa) in Javascript, you can combine built-in validators with the self-validation 'predicate' form like this: `'#number? _ > 0'`

```
# given this invalid input (string instead of number)
* def date = { month: '3' }
# this will pass
* match date == { month: '#? _ > 0' }
# but this 'combined form' will fail, which is what we want
# * match date == { month: '#number? _ > 0' }
```

Referring to the JSON root

You can actually refer to any JsonPath on the document via `$` and perform cross-field or conditional validations ! This example uses `contains` and the `#?` 'predicate' syntax, and situations where this comes in useful will be apparent when we discuss `match each`.

```
Given def temperature = { celsius: 100, fahrenheit: 212 }
Then match temperature == { celsius: '#number', fahrenheit: '#? _ ==
$.celsius * 1.8 + 32' }
# when validation logic is an 'equality' check, an embedded expression works
better
Then match temperature contains { fahrenheit: '#($.celsius * 1.8 + 32)' }
```

`match` for Text and Streams

```
# when the response is plain-text
Then match response == 'Health Check OK'
And match response != 'Error'

# when the response is a file (stream)
Then match response == read('test.pdf')

# incidentally, match and assert behave exactly the same way for strings
* def hello = 'Hello World!'
* match hello == 'Hello World!'
* assert hello == 'Hello World!'
```

Checking if a string is contained within another string is a very common need and `match` (name) `contains` works just like you'd expect:

```
* def hello = 'Hello World!'
* match hello contains 'World'
* match hello !contains 'blah'
```

`match header`

Since asserting against header values in the response is a common task - `match header` has a special meaning. It short-cuts to the pre-defined variable `responseHeaders` and reduces some

complexity - because strictly, HTTP headers are a 'multi-valued map' or a 'map of lists' - the Java-speak equivalent being `Map<String, List<String>>`.

```
# so after a http request
Then match header Content-Type == 'application/json'
# 'contains' works as well
Then match header Content-Type contains 'application'
```

Note the extra convenience where you don't have to enclose the LHS key in quotes.

You can always directly access the variable called `responseHeaders` if you wanted to do more checks, but you typically won't need to.

Matching Sub-Sets of JSON Keys and Arrays

`match contains`

JSON Keys

In some cases where the response JSON is wildly dynamic, you may want to only check for the existence of some keys. And `match` (name) `contains` is how you can do so:

```
* def foo = { bar: 1, baz: 'hello', ban: 'world' }

* match foo contains { bar: 1 }
* match foo contains { baz: 'hello' }
* match foo contains { bar:1, baz: 'hello' }
# this will fail
# * match foo == { bar:1, baz: 'hello' }
```

Also note that `match contains any` is possible for JSON objects as well as JSON arrays.

(not) `!contains`

It is sometimes useful to be able to check if a key-value-pair does **not** exist. This is possible by prefixing `contains` with a `!` (with no space in between).

```
* def foo = { bar: 1, baz: 'hello', ban: 'world' }
* match foo !contains { bar: 2 }
* match foo !contains { huh: '#notnull' }
```

Here's a reminder that the `#notpresent` marker can be mixed into an equality `match` (==) to assert that some keys exist and at the same time ensure that some keys do **not** exist:

```
* def foo = { a: 1 }
* match foo == { a: '#number', b: '#notpresent' }

# if b can be present (optional) but should always be null
* match foo == { a: '#number', b: '##null' }
```

The `!` (not) operator is especially useful for `contains` and JSON arrays.

```
* def foo = [1, 2, 3]
* match foo !contains 4
* match foo !contains [5, 6]
```

JSON Arrays

This is a good time to deep-dive into JsonPath, which is perfect for slicing and dicing JSON into manageable chunks. It is worth taking a few minutes to go through the documentation and examples here: JsonPath Examples.

Here are some example assertions performed while scraping a list of child elements out of the JSON below. Observe how you can `match` the result of a JsonPath expression with your expected data.

```
Given def cat =
"""
{
  name: 'Billie',
  kittens: [
    { id: 23, name: 'Bob' },
    { id: 42, name: 'Wild' }
  ]
}
"""
# normal 'equality' match. note the wildcard '*' in the JsonPath (returns an
array)
Then match cat.kittens[*].id == [23, 42]

# when inspecting a json array, 'contains' just checks if the expected items
exist
# and the size and order of the actual array does not matter
Then match cat.kittens[*].id contains 23
Then match cat.kittens[*].id contains [42]
Then match cat.kittens[*].id contains [23, 42]
Then match cat.kittens[*].id contains [42, 23]

# and yes, you can assert against nested objects within JSON arrays !
Then match cat.kittens contains [{ id: 42, name: 'Wild' }, { id: 23, name:
'Bob' }]

# ... and even ignore fields at the same time !
Then match cat.kittens contains { id: 42, name: '#string' }
```

It is worth mentioning that to do the equivalent of the last line in Java, you would typically have to traverse 2 Java Objects, one of which is within a list, and you would have to check for nulls as well.

When you use Karate, all your data assertions can be done in pure JSON and without needing a thick forest of companion Java objects. And when you `read` your JSON objects from (re-usable)

files, even complex response payload assertions can be accomplished in just a single line of Karate-script.

Refer to this case study for how dramatic the reduction of lines of code can be.

`match contains only`

For those cases where you need to assert that **all** array elements are present but in **any order** you can do this:

```
* def data = { foo: [1, 2, 3] }
* match data.foo contains 1
* match data.foo contains [2]
* match data.foo contains [3, 2]
* match data.foo contains only [3, 2, 1]
* match data.foo contains only [2, 3, 1]
# this will fail
# * match data.foo contains only [2, 3]
```

`match contains any`

To assert that **any** of the given array elements are present.

```
* def data = { foo: [1, 2, 3] }
* match data.foo contains any [9, 2, 8]
```

And this happens to work as expected for JSON object keys as well:

```
* def data = { a: 1, b: 'x' }
* match data contains any { b: 'x', c: true }
```

Validate every element in a JSON array

`match each`

The `match` keyword can be made to iterate over all elements in a JSON array using the `each` modifier. Here's how it works:

```
* def data = { foo: [{ bar: 1, baz: 'a' }, { bar: 2, baz: 'b' }, { bar: 3, baz: 'c' }]}

* match each data.foo == { bar: '#number', baz: '#string' }

# and you can use 'contains' the way you'd expect
* match each data.foo contains { bar: '#number' }
* match each data.foo contains { bar: '#? _ != 4' }

# some more examples of validation macros
* match each data.foo contains { baz: "#? _ != 'z'" }
* def isAbc = function(x) { return x == 'a' || x == 'b' || x == 'c' }
* match each data.foo contains { baz: '#? isAbc(_)' }
```

Here is a contrived example that uses `match each`, `contains` and the `#?` 'predicate' marker to validate that the value of `totalPrice` is always equal to the `roomPrice` of the first item in the `roomInformation` array.

```
Given def json =
"""
{
  "hotels": [
    { "roomInformation": [{ "roomPrice": 618.4 }], "totalPrice": 618.4  },
    { "roomInformation": [{ "roomPrice": 679.79}], "totalPrice": 679.79 }
  ]
}
"""
Then match each json.hotels contains { totalPrice: '#? _ ==
_$.roomInformation[0].roomPrice' }
# when validation logic is an 'equality' check, an embedded expression works
better
Then match each json.hotels contains { totalPrice:
'#(_$.roomInformation[0].roomPrice)' }
```

Referring to self

While `$` always refers to the JSON 'root', note the use of `_$` above to represent the 'current' node of a `match each` iteration. Here is a recap of symbols that can be used in JSON embedded expressions:

Symbol	Evaluates To
`$`	The 'root' of the JSON document in scope
`_`	The value of 'self'
`_$`	The 'parent' of 'self' or 'current' item in the list, relevant when using `match each`

There is a shortcut for `match each` explained in the next section that can be quite useful, especially for 'in-line' schema-like validations.

Schema Validation

Karate provides a far more simpler and more powerful way than JSON-schema to validate the stucture of a given payload. You can even mix domain and conditional validations and perform all assertions in a single step.

But first, a special short-cut for array validation needs to be introduced:

```
* def foo = ['bar', 'baz']

# should be an array
* match foo == '#[]'

# should be an array of size 2
* match foo == '#[2]'
```

```
# should be an array of strings with size 2
* match foo == '#[2] #string'

# each array element should have a 'length' property with value 3
* match foo == '#[]? _.length == 3'

# should be an array of strings each of length 3
* match foo == '#[] #string? _.length == 3'

# should be null or an array of strings
* match foo == '##[] #string'
```

This 'in-line' short-cut for validating JSON arrays is similar to how `match each` works. So now, complex payloads (that include arrays) can easily be validated in one step by combining validation markers like so:

```
* def oddSchema = { price: '#string', status: '#? _ < 3', ck: '##number',
name: '#regex[0-9X]' }
* def isValidTime = read('time-validator.js')
When method get
Then match response ==
"""
{
  id: '#regex[0-9]+',
  count: '#number',
  odd: '#(oddSchema)',
  data: {
    countryId: '#number',
    countryName: '#string',
    leagueName: '##string',
    status: '#number? _ >= 0',
    sportName: '#string',
    time: '#? isValidTime(_)'
  },
  odds: '#[] oddSchema'
}
"""
```

Especially note the re-use of the `oddSchema` both as an embedded-expression and as an array validation (on the last line).

And you can perform conditional / cross-field validations and even business-logic validations at the same time.

```
# optional (can be null) and if present should be an array of size greater
than zero
* match $.odds == '##[_ > 0]'

# should be an array of size equal to $.count
* match $.odds == '#[$.count]'

# use a predicate function to validate each array element
* def isValidOdd = function(o){ return o.name.length == 1 }
```

```
* match $.odds == '#[]? isValidOdd(_)'
```

Refer to this for the complete example: `schema-like.feature`

And there is another example in the karate-demos: `schema.feature` where you can compare Karate's approach with an actual JSON-schema example. You can also find a nice visual comparison and explanation here.

`contains` short-cuts

Especially when payloads are complex (or highly dynamic), it may be more practical to use `contains` semantics. Karate has the following short-cut symbols designed to be mixed into `embedded expressions`:

Symbol	Means
`^`	`contains`
`^^`	`contains only`
`^*`	`contains any`
`!^`	`not contains`

Here'a table of the alternative 'in-line' forms compared with the 'standard' form. Note that the short-cut forms on the right-side of the table mostly resolve to 'equality' (==) matches, which enables them to be 'in-lined' into a *full* (single-step) payload `match`, using embedded expressions.

```
* def actual = [{ a: 1, b: 'x' }, { a: 2, b: 'y' }]

* def schema = { a: '#number', b: '#string' }
* def partSchema = { a: '#number' }
* def badSchema = { c: '#boolean' }
* def mixSchema = { a: '#number', c: '#boolean' }

* def shuffled = [{ a: 2, b: 'y' }, { b: 'x', a: 1 }]
* def first = { a: 1, b: 'x' }
* def part = { a: 1 }
* def mix = { b: 'y', c: true }
* def other = [{ a: 3, b: 'u' }, { a: 4, b: 'v' }]
* def some = [{ a: 1, b: 'x' }, { a: 5, b: 'w' }]
```

Standard form	In-li
`* match actual[0] == schema`	* ma
`* match actual[0] contains partSchema`	* ma
`* match actual[0] contains any mixSchema`	* ma
`* match actual[0] !contains badSchema`	* ma
`* match each actual == schema`	* ma
`* match each actual contains partSchema`	* ma
`* match each actual contains any mixSchema`	* ma
`* match each actual !contains badSchema`	* ma
`* match actual contains only shuffled`	* ma
`* match actual contains first`	* ma
`* match actual contains any some`	* ma
`* match actual !contains other`	* ma
`* match actual contains '#(^part)'`	

A very useful capability is to be able to check that an array `contains` an object that `contains` the provided *sub-set* of keys instead of having to specify the *complete* JSON - which can get really cumbersome for large objects. This turns out to be very useful in practice, and this particular `match` *jsonArray* `contains '#(^`*partialObject*`)'` form has no 'in-line' equivalent (see the third-from-last row above).

The last row in the table is a little different from the rest, and this short-cut form is the recommended way to validate the length of a JSON array. As a rule of thumb, prefer `match` over `assert`, because `match` failure messages are more detailed and descriptive.

In real-life tests, these are very useful when the order of items in arrays returned from the server are not guaranteed. You can easily assert that all expected elements are present, *even* in nested parts of your JSON - while doing a `match` on the *full* payload.

```
* def cat =
"""
{
  name: 'Billie',
  kittens: [
    { id: 23, name: 'Bob' },
    { id: 42, name: 'Wild' }
  ]
}
"""
* def expected = [{ id: 42, name: 'Wild' }, { id: 23, name: 'Bob' }]
* match cat == { name: 'Billie', kittens: '#(^^expected)' }
```

There's a lot going on in the last line above ! It validates the entire payload in one step and checks if the `kittens` array ***contains all*** the `expected` items but in ***any order***.

`get`

By now, it should be clear that JsonPath can be very useful for extracting JSON 'trees' out of a given object. The `get` keyword allows you to save the results of a JsonPath expression for later use - which is especially useful for dynamic data-driven testing.

```
* def cat =
"""
{
  name: 'Billie',
  kittens: [
    { id: 23, name: 'Bob' },
    { id: 42, name: 'Wild' }
  ]
}
"""
* def kitnums = get cat.kittens[*].id
* match kitnums == [23, 42]
* def kitnames = get cat $.kittens[*].name
```

```
* match kitnames == ['Bob', 'Wild']
```

`get` short-cut

The 'short cut' `$variableName` form is also supported. Refer to JsonPath short-cuts for a detailed explanation. So the above could be re-written as follows:

```
* def kitnums = $cat.kittens[*].id
* match kitnums == [23, 42]
* def kitnames = $cat.kittens[*].name
* match kitnames == ['Bob', 'Wild']
```

It is worth repeating that the above can be condensed into 2 lines. Note that since only JsonPath is expected on the left-hand-side of the `==` sign of a `match` statement, you don't need to prefix the variable reference with `$`:

```
* match cat.kittens[*].id == [23, 42]
* match cat.kittens[*].name == ['Bob', 'Wild']

# if you prefer using 'pure' JsonPath, you can do this
* match cat $.kittens[*].id == [23, 42]
* match cat $.kittens[*].name == ['Bob', 'Wild']
```

`get` plus index

A convenience that the `get` syntax supports (but not the `$` short-cut form) is to return a single element if the right-hand-side evaluates to a list-like result (e.g. a JSON array). This is useful because the moment you use a wildcard `[*]` (or search filter) in JsonPath, you get a list back - even though typically you may be interested in only the first item.

```
* def actual = 23

# so instead of this
* def kitnums = get cat.kittens[*].id
* match actual == kitnums[0]

# you can do this in one line
* match actual == get[0] cat.kittens[*].id
```

JsonPath filters

JsonPath filter expressions are very useful for extracting elements that meet some filter criteria out of arrays.

```
* def cat =
"""
{
  name: 'Billie',
  kittens: [
    { id: 23, name: 'Bob' },
```

```
    { id: 42, name: 'Wild' }
  ]
}
"""
# find single kitten where id == 23
* def bob = get[0] cat.kittens[?(@.id==23)]
* match bob.name == 'Bob'

# using the karate object if the expression is dynamic
* def temp = karate.jsonPath(cat, "$.kittens[?(@.name=='" + bob.name + "')]")
* match temp[0] == bob

# or alternatively
* def temp = karate.jsonPath(cat, "$.kittens[?(@.name=='" + bob.name +
"')]")[0]
* match temp == bob
```

You usually won't need this, but the second-last line above shows how the `karate` object can be used to evaluate a JsonPath if the filter expression depends on a variable.

XPath Functions

When handling XML, you sometimes need to call XPath functions, for example to get the count of a node-set. Any valid XPath expression is allowed on the left-hand-side of a `match` statement.

```
* def myXml =
"""
<records>
  <record index="1">a</record>
  <record index="2">b</record>
  <record index="3" foo="bar">c</record>
</records>
"""

* match foo count(/records//record) == 3
* match foo //record[@index=2] == 'b'
* match foo //record[@foo='bar'] == 'c'
```

Advanced XPath

Some XPath expressions return a list of nodes (instead of a single node). But since you can express a list of data-elements as a JSON array - even these XPath expressions can be used in `match` statements.

```
* def teachers =
"""
<teachers>
  <teacher department="science">
    <subject>math</subject>
    <subject>physics</subject>
  </teacher>
  <teacher department="arts">
```

```
    <subject>political education</subject>
    <subject>english</subject>
  </teacher>
</teachers>
"""
* match teachers //teacher[@department='science']/subject == ['math',
'physics']
```

You can refer to this file (which is part of the Karate test-suite) for more XML examples: `xml-and-xpath.feature`

Special Variables

These are 'built-in' variables, there are only a few and all of them give you access to the HTTP response.

`response`

After every HTTP call this variable is set with the response body, and is available until the next HTTP request over-writes it. You can easily assign the whole `response` (or just parts of it using Json-Path or XPath) to a variable, and use it in later steps.

The response is automatically available as a JSON, XML or String object depending on what the response contents are.

As a short-cut, when running JsonPath expressions - `$` represents the `response`. This has the advantage that you can use pure JsonPath and be more concise. For example:

```
# the three lines below are equivalent
Then match response $ == { name: 'Billie' }
Then match response == { name: 'Billie' }
Then match $ == { name: 'Billie' }

# the three lines below are equivalent
Then match response.name == 'Billie'
Then match response $.name == 'Billie'
Then match $.name == 'Billie'
```

And similarly for XML and XPath, '/' represents the `response`

```
# the four lines below are equivalent
Then match response / == <cat><name>Billie</name></cat>
Then match response/ == <cat><name>Billie</name></cat>
Then match response == <cat><name>Billie</name></cat>
Then match / == <cat><name>Billie</name></cat>

# the three lines below are equivalent
Then match response /cat/name == 'Billie'
Then match response/cat/name == 'Billie'
Then match /cat/name == 'Billie'
```

JsonPath short-cuts

The `$varName` form is used on the right-hand-side of Karate expressions and is *slightly* different from pure JsonPath expressions which always begin with `$.` or `$[`. Here is a summary of what the different 'shapes' mean in Karate:

Shape	Description
`$.bar`	Pure JsonPath equivalent of `$response.bar` where `response` is a JSON object
`$[0]`	Pure JsonPath equivalent of `$response[0]` where `response` is a JSON array
`$foo.bar`	Evaluates the JsonPath `$.bar` on the variable `foo` which is a JSON object or map-like
`$foo[0]`	Evaluates the JsonPath `$[0]` on the variable `foo` which is a JSON array or list-like

There is no need to prefix variable names with `$` on the left-hand-side of `match` statements because it is implied. You *can* if you want to, but since *only* JsonPath (on variables) is allowed here, Karate ignores the `$` and looks only at the variable name. None of the examples in the documentation use the `$varName` form on the LHS, and this is the recommended best-practice.

responseCookies

The `responseCookies` variable is set upon any HTTP response and is a map-like (or JSON-like) object. It can be easily inspected or used in expressions.

```
* assert responseCookies['my.key'].value == 'someValue'

# karate's unified data handling means that even 'match' works
* match responseCookies contains { time: '#notnull' }

# ... which means that checking if a cookie does NOT exist is a piece of cake
* match responseCookies !contains { blah: '#notnull' }

# save a response cookie for later use
* def time = responseCookies.time.value
```

As a convenience, cookies from the previous response are collected and passed as-is as part of the next HTTP request. This is what is normally expected and simulates a web-browser - which makes it easy to script things like HTML-form based authentication into test-flows. Refer to the documentation for `cookie` for details and how you can disable this if need be.

Each item within `responseCookies` is itself a 'map-like' object. Typically you would examine the `value` property as in the example above, but `domain` and `path` are also available.

responseHeaders

See also `match header` which is what you would normally need.

But if you need to use values in the response headers - they will be in a variable named `responseHeaders`. Note that it is a 'map of lists' so you will need to do things like this:

```
* def contentType = responseHeaders['Content-Type'][0]
```

And just as in the `responseCookies` example above, you can use `match` to run complex validations on the `responseHeaders`.

`responseStatus`

You would normally only need to use the `status` keyword. But if you really need to use the HTTP response code in an expression or save it for later, you can get it as an integer:

```
* def uploadStatusCode = responseStatus
```

`responseTime`

The response time (in milliseconds) for every HTTP request would be available in a variable called `responseTime`. You can use this to assert that the response was returned within the expected time like so:

```
When method post
Then status 201
And assert responseTime < 1000
```

HTTP Header Manipulation

`configure headers`

Custom header manipulation for every HTTP request is something that Karate makes very easy and pluggable. For every HTTP request made from Karate, the internal flow is as follows:

- did we `configure` the value of `headers` ?
- if so, is the configured value a JavaScript function ?
    - if so, a `call` is made to that function.
    - did the function invocation return a map-like (or JSON) object ?
        - all the key-value pairs are added to the HTTP headers.
- or is the configured value a JSON object ?
    - all the key-value pairs are added to the HTTP headers.

This makes setting up of complex authentication schemes for your test-flows really easy. It typically ends up being a one-liner that appears in the `Background` section at the start of your test-scripts. You can re-use the function you create across your whole project.

Here is an example JavaScript function that uses some variables in the context (which have been possibly set as the result of a sign-in) to build the `Authorization` header.

In the example below, note the use of the `karate` object for getting the value of a dynamic variable. This is preferred because it takes care of situations such as if the value is 'undefined' in JavaScript.

```
function() {
  var uuid = '' + java.util.UUID.randomUUID(); // convert to string
  var out = { // so now the txid_header would be a unique uuid for each
request
    txid_header: uuid,
    ip_header: '123.45.67.89', // hard coded here, but also can be as dynamic
as you want
  };
  var authString = '';
  var authToken = karate.get('authToken'); // use the 'karate' helper to do a
'safe' get of a 'dynamic' variable
  if (authToken) { // and if 'authToken' is not null ...
    authString = ',auth_type=MyAuthScheme'
        + ',auth_key=' + authToken.key
        + ',auth_user=' + authToken.userId
        + ',auth_project=' + authToken.projectId;
  }
  // the 'appId' variable here is expected to have been set via karate-
config.js (bootstrap init) and will never change
  out['Authorization'] = 'My_Auth app_id=' + appId + authString;
  return out;
}
```

Assuming the above code is in a file called `my-headers.js`, the next section on calling other feature files shows how it looks like in action at the beginning of a test script.

Notice how once the `authToken` variable is initialized, it is used by the above function to generate headers for every HTTP call made as part of the test flow.

If a few steps in your flow need to temporarily change (or completely bypass) the currently-set header-manipulation scheme, just update the `headers` configuration value or set it to `null` in the middle of a script.

The karate-demo has an example showing various ways to `configure` or set headers: `headers.feature`

Code Reuse / Common Routines

`call`

In any complex testing endeavor, you would find yourself needing 'common' code that needs to be re-used across multiple test scripts. A typical need would be to perform a 'sign in', or create a fresh user as a pre-requisite for the scenarios being tested.

There are two types of code that can be `call`-ed. `*.feature` files and JavaScript functions.

Calling other `*.feature` files

When you have a sequence of HTTP calls that need to be repeated for multiple test scripts, Karate allows you to treat a `*.feature` file as a re-usable unit. You can also pass parameters into the `*.feature` file being called, and extract variables out of the invocation result.

Here is an example of using the `call` keyword to invoke another feature file, loaded using the `read` function:

```
Feature: which makes a 'call' to another re-usable feature

Background:
* configure headers = read('classpath:my-headers.js')
* def signIn = call read('classpath:my-signin.feature') { username: 'john', password: 'secret' }
* def authToken = signIn.authToken

Scenario: some scenario
# main test steps
```

The contents of `my-signin.feature` are shown below. A few points to note:

- Karate creates a new 'context' for the feature file being invoked but passes along all variables and configuration. This means that all your config variables and `configure` settings would be available to use, for example `loginUrlBase` in the example below.
- When you use `def` in the 'called' feature, it will **not** over-write variables in the 'calling' feature (unless you explicitly choose to use shared scope). But note that JSON, XML, Map-like or List-like variables are 'passed by reference' which means that 'called' feature steps can *update* or 'mutate' them using the `set` keyword. Use the `copy` keyword to 'clone' a JSON or XML payload if needed, and refer to this example for more details: `copy-caller.feature`.
- You can add (or over-ride) variables by passing a call 'argument' as shown above. Only one JSON argument is allowed, but this does not limit you in any way as you can use any complex JSON structure. You can even initialize the JSON in a separate step and pass it by name, especially if it is complex. Observe how using JSON for parameter-passing makes things super-readable. In the 'called' feature, the argument can also be accessed using the built-in variable: `__arg`.
- **All** variables that were defined (using `def`) in the 'called' script would be returned as 'keys' within a JSON-like object. Note that this includes 'built-in' variables, which means that things like the last value of `response` would also be present. In the example above you can see that the JSON 'envelope' returned - is assigned to the variable named `signIn`. And then getting hold of any data that was generated by the 'called' script is as simple as accessing it by name, for example `signIn.authToken` as shown above. This design has the following advantages:
    - 'called' Karate scripts don't need to use any special keywords to 'return' data and can behave like 'normal' Karate tests in 'stand-alone' mode if needed

- the data 'return' mechanism is 'safe', there is no danger of the 'called' script over-writing any variables in the 'calling' (or parent) script (unless you use shared scope)
- the need to explicitly 'unpack' variables by name from the returned 'envelope' keeps things readable and maintainable in the 'caller' script

```
Feature: here are the contents of 'my-signin.feature'

Scenario:

Given url loginUrlBase
And request { userId: '#(username)', userPass: '#(password)' }
When method post
Then status 200
And def authToken = response

# second HTTP call, to get a list of 'projects'
Given path 'users', authToken.userId, 'projects'
When method get
Then status 200
# logic to 'choose' first project
And set authToken.projectId = response.projects[0].projectId;
```

The above example actually makes two HTTP requests - the first is a standard 'sign-in' POST and then (for illustrative purposes) another HTTP call (a GET) is made for retrieving a list of projects for the signed-in user, and the first one is 'selected' and added to the returned 'auth token' JSON object.

So you get the picture, any kind of complicated 'sign-in' flow can be scripted and re-used.

If the second HTTP call above expects headers to be set by `my-headers.js` - which in turn depends on the `authToken` variable being updated, you will need to duplicate the line `* configure headers = read('classpath:my-headers.js')` from the 'caller' feature here as well. The above example does **not** use shared scope, which means that the variables in the 'calling' (parent) feature are *not* shared by the 'called' `my-signin.feature`. The above example can be made more simpler with the use of `call` (or `callonce`) *without* a `def`-assignment to a variable, and is the recommended pattern for implementing re-usable authentication setup flows.

Do look at the documentation and example for `configure headers` also as it goes hand-in-hand with `call`. In the above example, the end-result of the `call` to `my-signin.feature` resulted in the `authToken` variable being initialized. Take a look at how the `configure headers` example uses the `authToken` variable.

Data-Driven Features

If the argument passed to the call of a `*.feature` file is a JSON array, something interesting happens. The feature is invoked for each item in the array. Each array element is expected to be a JSON object, and for each object - the behavior will be as described above.

But this time, the return value from the `call` step will be a JSON array of the same size as the input array. And each element of the returned array will be the 'envelope' of variables that resulted from each iteration where the `*.feature` got invoked.

Here is an example that combines the `table` keyword with calling a `*.feature`. Observe how the `get` shortcut is used to 'distill' the result array of variable 'envelopes' into an array consisting only of `response` payloads.

```
* table kittens
    | name   | age |
    | 'Bob'  |   2 |
    | 'Wild' |   1 |
    | 'Nyan' |   3 |

* def result = call read('cat-create.feature') kittens
* def created = $result[*].response
* match each created == { id: '#number', name: '#string', age: '#number' }
* match created[*].name contains only ['Bob', 'Wild', 'Nyan']
```

And here is how `cat-create.feature` could look like:

```
@ignore
Feature:

Scenario:

Given url someUrlFromConfig
And path 'cats'
And request { name: '#(name)', age: '#(age)' }
When method post
Then status 200
```

If you replace the `table` with perhaps a JavaScript function call that gets some JSON data from some data-source, you can imagine how you could go about dynamic data-driven testing.

Although it is just a few lines of code, take time to study the above example carefully. It is a great example of how to effectively use the unique combination of Cucumber and JsonPath that Karate provides.

Also look at the demo examples, especially `dynamic-params.feature` - to compare the above approach with how the Cucumber `Scenario Outline:` can be alternatively used for data-driven tests.

Built-in variables for `call`

Although all properties in the passed JSON-like argument are 'unpacked' into the current scope as separate 'named' variables, it sometimes makes sense to access the whole argument and this can be done via `__arg`. And if being called in a loop, a built-in variable called `__loop` will also be available that will hold the value of the current loop index. So you can do things like this: `*`

`def name = name + __loop` - or you can use the loop index value for looking up other values that may be in scope - in a data-driven style.

Variable	Refers To
`__arg`	the single `call` (or `callonce`) argument, will be `null` if there was none
`__loop`	the current iteration index if being called in a loop, will be `-1` if not

Refer to this demo feature for an example: `kitten-create.feature`

`copy`

For a `call` (or `callonce`) - payload / data structures (JSON, XML, Map-like or List-like) variables are 'passed by reference' which means that steps within the 'called' feature can update or 'mutate' them, for e.g. using the `set` keyword. This is actually the intent most of the time and is convenient. If you want to pass a 'clone' to a 'called' feature, you can do so using the rarely used `copy` keyword that works very similar to type conversion. This is best explained in the last scenario of this example: `copy-caller.feature`

Calling JavaScript Functions

Examples of defining and using JavaScript functions appear in earlier sections of this document. Being able to define and re-use JavaScript functions is a powerful capability of Karate. For example, you can:

- call re-usable functions that take complex data as an argument and return complex data that can be stored in a variable
- call and interoperate with Java code if needed
- share and re-use test utilities or 'helper' functionality across your organization

In real-life scripts, you would typically also use this capability of Karate to `configure headers` where the specified JavaScript function uses the variables that result from a sign in to manipulate headers for all subsequent HTTP requests. And it is worth mentioning that the Karate configuration 'bootstrap' routine is itself a JavaScript function.

Also refer to the `eval` keyword for a simpler way to execute arbitrary JavaScript that can be useful in some situations.

The `karate` object

A JavaScript function or expression at runtime has access to a utility object in a variable named: `karate`. This provides the following methods:

Operation	Description
`karate.set(name, value)`	sets the value of a variable (immediately), which may be

Operation	Description
	needed in case any other routines (such as the configured headers) depend on that variable
`karate.setXml(name, xmlString)`	rarely used, refer to the next example
`karate.set(name, path, value)`	only needed when you need to conditionally build payload elements, especially XML. This is best explained via an example, and it behaves the same way as the `set` keyword.
`karate.remove(name, path)`	similar to the above, again very rarely used - when needing to perform conditional removal of XML nodes. Behaves the same way as the `remove` keyword.
`karate.get(name)`	get the value of a variable by name (or JsonPath expression), if not found - this returns `null` which is easier to handle in JavaScript (than `undefined`)
`karate.jsonPath(json, expression)`	brings the power of JsonPath into Karate-JS, and you can find an example here.
`karate.read(filename)`	read from a file, behaves exactly like `read`
`karate.log(... args)`	log to the same logger (and log file) being used by the parent process, logging can be suppressed with `configure printEnabled` set to `false`
`karate.pretty(value)`	return a 'pretty-printed', nicely indented string representation of the JSON value, also see: `print`
`karate.prettyXml(value)`	return a 'pretty-printed', nicely indented string representation of the XML value, also see: `print`
`karate.env`	gets the value (read-only) of the environment property 'karate.env', and this is typically used for bootstrapping configuration
`karate.properties[key]`	get the value of any Java system-property by name, useful for advanced custom configuration
`karate.configure(key, value)`	does the same thing as the `configure` keyword, and a very useful example is to do `karate.configure('connectTimeout', 5000);` in `karate-config.js` - which has the 'global' effect of not wasting time if a connection cannot be established within 5 seconds
`karate.toBean(json, className)`	converts a JSON string or map-like object into a Java bean (or POJO), given the Java class name as the second argument, refer to this file for an example
`karate.call(fileName, [arg])`	invoke a `*.feature` file or a JavaScript function the same way that `call` works (with an optional solitary argument)
`karate.callSingle(fileName, [arg])`	like the above, but guranteed to run **only once** even across multiple features *and* parallel threads (recommended only for advanced users) - refer to this example: `karate-config.js` /

Operation	Description
	`headers-single.feature`
`karate.eval(expression)`	for really advanced needs, you can programmatically generate a snippet of JavaScript which can be evaluated at run-time, you can find an example here
`karate.tags`	for advanced users - scripts can introspect the tags that apply to the current scope, refer to this example: `tags.feature`
`karate.tagValues`	for even more advanced users - Karate natively supports tags in a `@name=val1,val2` format, and there is an inheritance mechanism where `Scenario` level tags can over-ride `Feature` level tags, refer to this example: `tags.feature`
`karate.prevRequest`	for advanced users, you can inspect the *actual* HTTP request after it happens, useful if you are writing a framework over Karate, refer to this example: `request.feature`
`karate.info`	within a test (or within the `afterScenario` function if configured) you can access metadata such as the `Scenario` name, refer to this example: `hooks.feature`

JS function argument rules for `call`

When using `call` (or `callonce`), only one argument is allowed. But this does not limit you in any way, because similar to how you can call `*.feature files`, you can pass a whole JSON object as the argument. In the case of the `call` of a JavaScript function, you can also pass a JSON array or a primitive (string, number, boolean) as the solitary argument, and the function implementation is expected to handle whatever is passed.

Instead of using `call` (or `callonce`) you are always free to call JavaScript functions 'normally' and then you can use more than one argument.

```
* def adder = function(a, b){ return a + b }
* assert adder(1, 2) == 3
```

Return types

Naturally, only one value can be returned. But again, you can return a JSON object. There are two things that can happen to the returned value.

Either - it can be assigned to a variable like so.

```
* def returnValue = call myFunction
```

Or - if a `call` is made without an assignment, and if the function returns a map-like object, it will add each key-value pair returned as a new variable into the execution context.

```
# while this looks innocent ...
```

```
# ... behind the scenes, it could be creating (or over-writing) a bunch of
variables !
* call someFunction
```

While this sounds dangerous and should be used with care (and limits readability), the reason this feature exists is to quickly set (or over-write) a bunch of config variables when needed. In fact, this is the mechanism used when `karate-config.js` is processed on start-up.

Shared Scope

This behavior where all key-value pairs in the returned map-like object get automatically added as variables - applies to the calling of `*.feature` files as well. In other words, when `call` or `callonce` is used without a `def`, the 'called' script not only shares all variables (and `configure` settings) but can update the shared execution context. This is very useful to boil-down those 'common' steps that you may have to perform at the start of multiple test-scripts - into one-liners. But use wisely, because called scripts will now over-write variables that may have been already defined.

```
* def config = { user: 'john', password: 'secret' }
# this next line may perform many steps and result in multiple variables set
for the rest of the script
* call read('classpath:common-setup.feature') config
```

You can use `callonce` instead of `call` within the `Background` in case you have multiple `Scenario` sections or `Examples`. Note the 'inline' use of the read function as a short-cut above. This applies to JS functions as well:

```
* call read('my-function.js')
```

These heavily commented demo examples can help you understand 'shared scope' better, and are designed to get you started with creating re-usable 'sign-in' or authentication flows:

Scope	Caller Feature	Called Feature
Isolated	`call-isolated-headers.feature`	`common-multiple.feature`
Shared	`call-updates-config.feature`	`common.feature`

Once you get comfortable with Karate, you can consider moving your authentication flow into a 'global' one-time flow using `karate.callSingle()`, think of it as '`callonce` on steroids'.

HTTP Basic Authentication Example

This should make it clear why Karate does not provide 'out of the box' support for any particular HTTP authentication scheme. Things are designed so that you can plug-in what you need, without needing to compile Java code. You get to choose how to manage your environment-specific configuration values such as user-names and passwords.

First the JavaScript file, `basic-auth.js`:

```
function(creds) {
  var temp = creds.username + ':' + creds.password;
  var Base64 = Java.type('java.util.Base64');
  var encoded = Base64.getEncoder().encodeToString(temp.bytes);
  return 'Basic ' + encoded;
}
```

And here's how it works in a test-script using the `header` keyword.

```
* header Authorization = call read('basic-auth.js') { username: 'john',
password: 'secret' }
```

You can set this up for all subsequent requests or dynamically generate headers for each HTTP request if you `configure headers`.

Calling Java

There are examples of calling JVM classes in the section on Java Interop and in the file-upload demo. Also look at the section on commonly needed utilities for more ideas.

Calling any Java code is that easy. Given this custom, user-defined Java class:

```
package com.mycompany;

import java.util.HashMap;
import java.util.Map;

public class JavaDemo {

    public Map<String, Object> doWork(String fromJs) {
        Map<String, Object> map = new HashMap<>();
        map.put("someKey", "hello " + fromJs);
        return map;
    }

    public static String doWorkStatic(String fromJs) {
        return "hello " + fromJs;
    }

}
```

This is how it can be called from a test-script, and yes, even static methods can be invoked:

```
* def doWork =
"""
function(arg) {
  var JavaDemo = Java.type('com.mycompany.JavaDemo');
  var jd = new JavaDemo();
  return jd.doWork(arg);
}
"""
# in this case the solitary 'call' argument is of type string
* def result = call doWork 'world'
```

```
* match result == { someKey: 'hello world' }

# using a static method - observe how java interop is truly seamless !
* def JavaDemo = Java.type('com.mycompany.JavaDemo')
* def result = JavaDemo.doWorkStatic('world')
* assert result == 'hello world'
```

Note that JSON gets auto-converted to `Map` (or `List`) when making the cross-over to Java. Refer to the `cats-java.feature` demo for an example.

Another great example is `dogs.feature` - which actually makes JDBC (database) calls, and since the data returned from the Java code is JSON, the last section of the test is able to use `match` *very* effectively for data assertions.

`callonce`

Cucumber has a limitation where `Background` steps are re-run for every `Scenario`. And if you have a `Scenario Outline`, this happens for *every* row in the `Examples`. This is a problem especially for expensive, time-consuming HTTP calls, and this has been an open issue for a long time.

Karate's `callonce` keyword behaves exactly like `call` but is guaranteed to execute only once. The results of the first call are cached, and any future calls will simply return the cached result instead of executing the JavaScript function (or feature) again and again.

This does require you to move 'set-up' into a separate `*.feature` (or JavaScript) file. But this totally makes sense for things not part of the 'main' test flow and which typically need to be re-usable anyway.

So when you use the combination of `callonce` in a `Background`, you can indeed get the same effect as using a `@BeforeClass` annotation, and you can find examples in the karate-demo, such as this one: `callonce.feature`.

Recommended only for experienced users - `karate.callSingle()` is a way to invoke a feature or function 'globally' only once.

`eval`

This is for evaluating arbitrary JavaScript and you are advised to use this only as a last resort ! Conditional logic is not recommended especially within test scripts because tests should be deterministic.

There are a few situations where this comes in handy:

- you *really* don't need to assign a result to a variable
- statements in the `if` form (also see conditional logic)

- 'one-off' logic (or Java interop) where you don't need the 'ceremony' of a re-usable function

```
# just perform an action, we don't care about saving the result
* eval myJavaScriptFunction()

# do something only if a condition is true
* eval if (zone == 'zone1') karate.set('temp', 'after')

# you can use multiple lines of JavaScript if needed
* eval
"""
var foo = function(v){ return v * v };
var nums = [0, 1, 2, 3, 4];
var squares = [];
for (var n in nums) {
  squares.push(foo(n));
}
karate.set('temp', squares);
"""
* match temp == [0, 1, 4, 9, 16]
```

Advanced / Tricks

Polling

Waiting or performing a 'sleep' until a certain condition is met is a common need, and this demo example should get you up and running: `polling.feature`.

Conditional Logic

The keywords `Given` `When` `Then` are only for decoration and should not be thought of as similar to an `if - then - else` statement. And as a testing framework, Karate discourages tests that give different results on every run.

That said, if you really need to implement 'conditional' checks, this can be one pattern:

```
* def filename = (zone == 'zone1' ? 'test1.feature' : 'test2.feature')
* def result = call read(filename)
```

And this is another, using `karate.call()`. Here we want to `call` a file only if a condition is satisfied:

```
* def result = (responseStatus == 404 ? {} : karate.call('delete-
user.feature'))
```

Or if we don't care about the result, we can use `eval`:

```
* eval if (responseStatus == 404) karate.call('delete-user.feature')
```

And this may give you more ideas. You can always use a JavaScript function or call Java for more complex logic.

```
* def expected = (zone == 'zone1' ? { foo: '#string' } : { bar: '#number' })
* match response == expected
```

Also refer to polling for more ideas.

Commonly Needed Utilities

Since it is so easy to dive into Java-interop, Karate does not include any random-number functions, uuid generator or date / time utilities out of the box. You simply roll your own.

Here is an example of how to get the current date, and formatted the way you want:

```
* def getDate =
"""
function() {
  var SimpleDateFormat = Java.type('java.text.SimpleDateFormat');
  var sdf = new SimpleDateFormat('yyyy/MM/dd');
  var date = new java.util.Date();
  return sdf.format(date);
}
"""

* def temp = getDate()
* print temp
```

And the above will result in something like this being logged: `[print] 2017/10/16`.

Here below are a few more common examples:

Utility	Recipe
System Time	`function(){ return java.lang.System.currentTimeMillis() }`
UUID	`function(){ return java.util.UUID.randomUUID() + '' }`

The above are good enough for the purposes of random string generation for most situations.

GraphQL / RegEx replacement example

As a demonstration of Karate's power and flexibility, here is an example that reads a GraphQL string (which could be from a file) and manipulates it to build custom dynamic queries and filter criteria.

Here we have this JavaScript utlity function `replacer.js` that uses a regular-expression to replace-inject a criteria expression into the right place, given a GraphQL query.

```
function(args) {
  var query = args.query;
  var regex = new RegExp('\\s' + args.field + '\\s*{'); // the RegExp object
is standard JavaScript
  return query.replace(regex, ' ' + args.field + '(' + args.criteria + ')
{');
}
```

Once the function is declared, observe how calling it and performing the replacement is an elegant one-liner.

```
* def replacer = read('replacer.js')

# this 'base GraphQL query' would also likely be read from a file in real-
life
* def query = 'query q { company { taxAgencies { edges { node { id, name } }
} } }'

# the next line is where the criteria is injected using the regex function
* def query = call replacer { query: '#(query)', field: 'taxAgencies',
criteria: 'first: 5' }

# and here is the result of the 'replace'
* assert query == 'query q { company { taxAgencies(first: 5) { edges { node {
id, name } } } } }'

Given request { query: '#(query)' }
And header Accept = 'application/json'
When method post
Then status 200

* def agencies = $.data.company.taxAgencies.edges
* match agencies[0].node == { id: '#uuid', name: 'John Smith' }
```

The example above is more for demonstration purposes and it is better practice to use GraphQL variables for dynamic queries. This example is a good reference: `graphql.feature`. Also the `replace` keyword may be all you need for simple text placeholder substitution.

Cucumber Tags

Cucumber has a great way to sprinkle meta-data into test-scripts - which gives you some interesting options when running tests in bulk. The most common use-case would be to partition your tests into 'smoke', 'regression' and the like - which enables being able to selectively execute a sub-set of tests.

The documentation on how to run tests via the command line has an example of how to use tags to decide which tests to *not* run (or ignore). The Cucumber wiki has more information on tags. Also see `first.feature` and `second.feature` in the demos.

For advanced users, Karate supports being able to query for tags within a test, and even tags in a `@name=value` form. Refer to the `karate.tags` and `karate.tagValues` methods on the Karate JS object.

Dynamic Port Numbers

In situations where you start an (embedded) application server as part of the test set-up phase, a typical challenge is that the HTTP port may be determined at run-time. So how can you get this value injected into the Karate configuration ?

It so happens that the `karate` object has a field called `properties` which can read a Java system-property by name like this: `karate.properties['myName']`. Since the `karate` object is injected within `karate-config.js` on start-up, it is a simple and effective way for other processes within the same JVM to pass configuration values to Karate at run-time. Refer to the 'demo' `karate-config.js` for an example and how the `demo.server.port` system-property is set-up in the test runner: `TestBase.java`.

Java API

It should be clear now that Karate provides a super-simple way to make HTTP requests compared to how you would have done so in Java. It is also possible to invoke a feature file via a Java API which can be very useful in some test-automation situations.

A common use case is to mix API-calls into a larger test-suite, for example a Selenium or WebDriver UI test. So you can use Karate to set-up data via API calls, then run the UI test-automation, and finally again use Karate to assert that the system-state is as expected. Note that you can even include calls to a database from Karate using Java interop. And this example may make it clear why using Karate itself to drive even your UI-tests may be a good idea.

There are two static methods in `com.intuit.karate.cucumber.CucumberRunner` (`runFeature()` and `runClasspathFeature()`) which are best explained in this demo unit-test: `JavaApiTest.java`.

You can optionally pass in variable values or over-ride config via a `HashMap` or leave the second-last argument as `null`. The variable state after feature execution would be returned as a `Map<String, Object>`. The last `boolean` argument is whether the `karate-config.js` should be processed or not. Refer to the documentation on type-conversion to make sure you can 'unpack' data returned from Karate correctly, especially when dealing with XML.

Hooks

If you are looking for Cucumber 'hooks' Karate does not support them, mainly because they depend on Java code, which goes against the Karate Way™.

Instead, Karate gives you all you need as part of the syntax. Here is a summary:

To Run Some Code	How
Before *everything* (or 'globally' once)	Use `karate.callSingle()` in `karate-config.js`. Only recommended for advanced users, but this guarantees a routine is run only once, *even* when running tests in parallel.
Before every `Scenario`	Use the `Background`
Once (or at the start of) every `Feature`	Use a `callonce` in the `Background`. The advantage is that you can set up variables (using `def` if needed) which can be used in all `Scenario`-s within that `Feature`.
After every `Scenario`	`configure afterScenario` (see example)
At the end of the `Feature`	`configure afterFeature` (see example)

Data Driven Tests

The Cucumber Way

Cucumber has a concept of Scenario Outlines where you can re-use a set of data-driven steps and assertions, and the data can be declared in a very user-friendly fashion. Observe the usage of `Scenario Outline:` instead of `Scenario:`, and the new `Examples:` section.

You should take a minute to compare this with the exact same example implemented in REST-assured and TestNG.

```
Feature: karate answers 2

Background:
* url 'http://localhost:8080'

Scenario Outline: given circuit name, validate country

Given path 'api/f1/circuits/<name>.json'
When method get
Then match $.MRData.CircuitTable.Circuits[0].Location.country == '<country>'

Examples:
| name    | country  |
| monza   | Italy    |
| spa     | Belgium  |
| sepang | Malaysia |

Scenario Outline: given race number, validate number of pitstops for Max
Verstappen in 2015
```

```
Given path 'api/f1/2015/<race>/drivers/max_verstappen/pitstops.json'
When method get
Then assert response.MRData.RaceTable.Races[0].PitStops.length == <stops>

Examples:
| race | stops |
| 1    | 1     |
| 2    | 3     |
| 3    | 2     |
| 4    | 2     |
```

This is great for testing boundary conditions against a single end-point, with the added bonus that your test becomes even more readable. This approach can certainly enable product-owners or domain-experts who are not programmer-folk, to review, and even collaborate on test-scenarios and scripts.

For an advanced example, see: `examples.feature`.

The Karate Way

The limitation of the Cucumber `Scenario Outline:` is that the number of rows in the `Examples:` is fixed. But take a look at how Karate can loop over a `*.feature` file for each object in a JSON array - which gives you dynamic data-driven testing, if you need it. For advanced examples, refer to some of the scenarios within this demo: `dynamic-params.feature`.